INTRODUCTION TO PROGRAMMING IN BASIC

RATE TRAINING MANUAL AND OFFICER-ENLISTED CORRESPONDENCE COURSE

NAVEDTRA 10079-2

Although the words "he," "him," and "his" are used sparingly in this manual to enhance communication, they are not intended to be gender driven nor to affront or discriminate against anyone reading *Introduction to Programming In BASIC*, NAVEDTRA 10079-2.

Naval Education And Training Program Development Center

INTRODUCTION TO PROGRAMMING IN BASIC

NAVEDTRA 10079-2

1983 Edition Written by
DPC Fred H. McGee, III

PREFACE

BASIC (Beginners All-Purpose Symbolic Instruction Code) is a high-level computer programming language designed for time-sharing. Its conversational nature makes communicating with a computer natural, simple, and straightforward. It is based on near-English words and mathematical expressions. In 1978, the American National Standards Institute issued a standard for minimal BASIC (ANSI X3.60-1978) which was adopted by the National Bureau of Standards as a Federal Information Processing Standard (FIPS PUB 68) in 1980. Where applicable, this standard was used as a guide in developing this text. However, because of the wide variety of implementations of BASIC, it was not possible to cover all the different implementations. Not all instructions are available on all computer systems, and the instructions and syntax may vary from the standard or the presentation in this text. The BASIC user's manual that accompanies the computer to be used must be consulted to determine any deviations.

This text is designed for individual study and requires no prior programming experience. For personnel with computer operations experience, it is intended to provide a transition from operations into programming. For those with experience in other programming languages, it introduces the BASIC language and its syntax. For others who have micro/mini computers available, this text provides an introduction to programming in BASIC. Each chapter includes end-of-chapter exercises with suggested solutions.

The OCC-ECC (Officer and Enlisted Correspondence Course) designed for use with this text follows the Index. It consists of assignments with a set of answer sheets. The assignments include learning objectives and supporting questions designed to emphasize key points covered in the text.

This text and the associated OCC-ECC were prepared by the Naval Education and Training Program Development Center, Pensacola, Florida, for the Chief of Naval Education and Training. Special appreciation is given to the following commands for their reviews during preparation of this text: Data Processing Technician "A" School, San Diego, California; Commander Mine Warfare Command, Charleston, South Carolina; Naval Regional Data Automation Center, Pensacola, Florida; Enlisted Personnel Management Center, New Orleans, Louisiana; COMSUBLANT, ADP Officer, Norfolk, Virginia; Computer Sciences School, Quantico, Virginia; and Statistical Analysis Division, NETPDC, Pensacola, Florida.

Your suggestions and comments are invited. Address them to NETPDC, Code PD-6, Pensacola, FL 32509.

1983 Edition
Reprinted 1984
Reprinted 1985

Stock Ordering No.
0502-LP-050-3970

Published by
NAVAL EDUCATION AND TRAINING
PROGRAM DEVELOPMENT CENTER

UNITED STATES
GOVERNMENT PRINTING OFFICE
WASHINGTON, D.C.: 1983

THE UNITED STATES NAVY

GUARDIAN OF OUR COUNTRY

The United States Navy is responsible for maintaining control of the sea and is a ready force on watch at home and overseas, capable of strong action to preserve the peace or of instant offensive action to win in war.

It is upon the maintenance of this control that our country's glorious future depends; the United States Navy exists to make it so.

WE SERVE WITH HONOR

Tradition, valor, and victory are the Navy's heritage from the past. To these may be added dedication, discipline, and vigilance as the watchwords of the present and the future.

At home or on distant stations we serve with pride, confident in the respect of our country, our shipmates, and our families.

Our responsibilities sober us; our adversities strengthen us.

Service to God and Country is our special privilege. We serve with honor.

THE FUTURE OF THE NAVY

The Navy will always employ new weapons, new techniques, and greater power to protect and defend the United States on the sea, under the sea, and in the air.

Now and in the future, control of the sea gives the United States her greatest advantage for the maintenance of peace and for victory in war.

Mobility, surprise, dispersal, and offensive power are the keynotes of the new Navy. The roots of the Navy lie in a strong belief in the future, in continued dedication to our tasks, and in reflection on our heritage from the past.

Never have our opportunities and our responsibilities been greater.

CONTENTS

IMPORTANT NOTE

Before writing any program for a specific computer, consult the BASIC user's manual for that computer. The BASIC instructions, words, and punctuation are NOT the same on all computers.

CHAPTER 1

INTRODUCTION TO PROGRAMMING

Problem Solving Concepts, Flowcharting, and Programming Languages

It is the intent of this manual to provide an introduction to computer programming, and to the programming language, BASIC (Beginners All-Purpose Symbolic Instruction Code). BASIC is a popular programming language, especially for new programmers and casual computer users. Its conversational nature makes communicating with computers natural, simple, and straightforward. Its use of near English words and mathematical expressions gives the coding a familiar appearance. Also, its original design, to teach the casual user how to program, makes it a good language to learn first.

For those of you with computer operations experience, this course is intended to provide a transition from operations into programming. It will introduce concepts of programming that apply to other languages as well as BASIC.

For those of you with programming experience, it will provide a review of programming concepts and introduce the capabilities of BASIC and its syntax.

For those not in data processing, who have a computer available, it will provide an introduction to a programming language available on most computers.

When you complete the course, you should understand the capabilities and syntax of the BASIC language and be able to write a program.

OVERVIEW OF PROGRAMMING

Before learning to program in the language, BASIC, it is helpful to establish some context for the productive part of the entire programming effort. This context comprises the understanding and agreement that there are four fundamental and discrete steps involved in solving a problem on a computer.

The four steps are:

1. State, analyze, and define the problem.
2. Develop the program logic and prepare a program flowchart or decision table.
3. Code the program, prepare the code in machine readable form, prepare test data, and perform debug and test runs.
4. Complete the documentation and prepare operator procedures for implementation and production.

Figure 1-1 depicts the evolution of a program.

Programming can be complicated, and advance preparation is required before you can actually start to write or code the program. The first two steps, problem understanding/definition and flowcharting, fall into the advance planning phase of programming. It is important at this point to develop

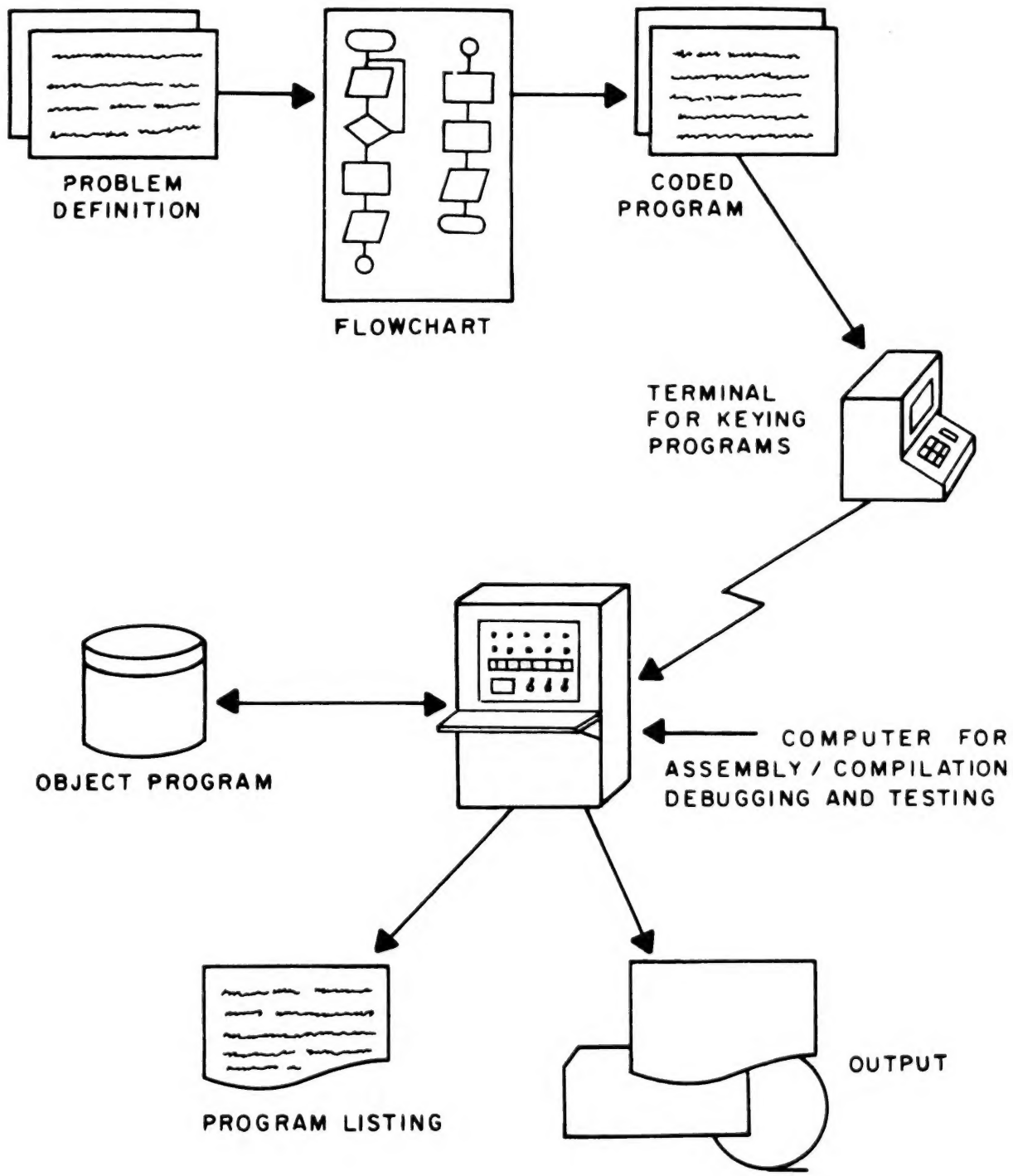


Figure 1-1.—Evolution of a Program.

correct habits and procedures, since this will prevent later difficulties in program preparation.

Whether you are working with a systems analyst, a customer, or solving a problem of your own, it is extremely important that you have a thorough understanding of the problem.

Every aspect of the problem must be defined:

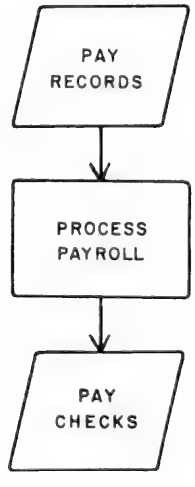
- What is the problem?
- What information (or data) is needed?
- Where and how will the information be obtained?
- What is the desired output?

Starting with only a portion of the information, or an incomplete definition, will result in having to constantly alter what has been done to accommodate the additional facts as they become available. It is easier and more efficient to begin programming after **all** of the necessary information is understood. Once you have a thorough understanding of the problem, the next step is flowcharting.

FLOWCHARTING

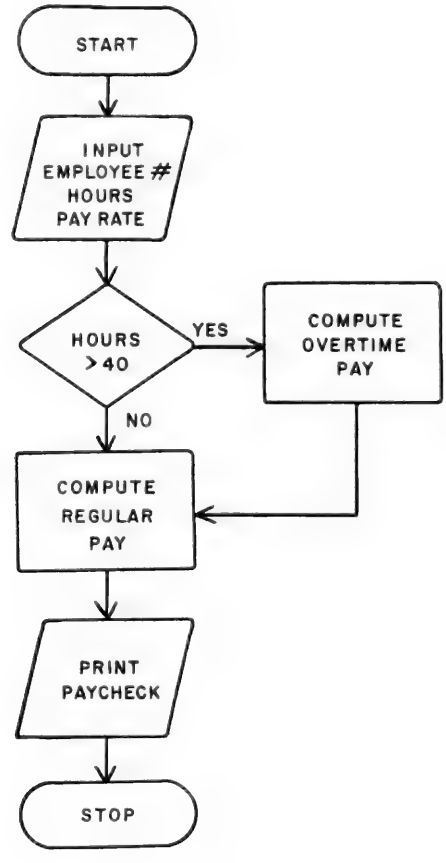
Flowcharting is one method of pictorially representing a procedural (step-by-step) solution to a problem before you actually start to write the computer instructions required to produce the desired results. Flowcharts use different shaped symbols connected by one-way arrows to represent operations, data, flow, equipment, and so forth.

There are two types of flowcharts, system (data) flowcharts and programming flowcharts. A system (data) flowchart defines the major phases of the processing, as well as the various data media used. It shows the relationship of numerous jobs that make up an entire system. In the system (data) flowchart, an entire program run or phase is always represented by a single processing symbol, together with the input/output symbols showing the path of data through a problem solution. For example:



System Flowchart

The second type of flowchart, and the one we'll use in this manual is the programming flowchart. It is constructed by the programmer to represent the sequence of operations the computer is to perform to solve a specific problem. It graphically describes what is to take place in the program. It displays specific operations and decisions, and their sequence within the program. For example:



Programming Flowchart

Tools of Flowcharting

Flowcharting has been defined, and two different types of flowcharts discussed. We will now take a look at the tools used in flowcharting. These tools are the fundamental symbols, graphic symbols, flowcharting template, and the flowcharting worksheet.

FUNDAMENTAL SYMBOLS.—To construct a flowchart, it is first necessary to know the symbols and their related meanings. They are standard for the military, as directed by Department of the Navy Automated Data Systems Documentation Standards, SECNAVINST 5233.1 (Series).

Symbols are used to represent functions. These fundamental functions are processing, decision, input/output, terminal, flow lines and connector symbol. All flowcharts may be initially constructed using only these fundamental symbols as a rough outline to work from. Each symbol corresponds to one of the functions of a computer and specifies the instruction(s) to be performed by the computer. The contents of these symbols are called statements. Samples of these fundamental symbols, definitions, examples, and explanations of their uses are shown in figure 1-2.

GRAPHIC SYMBOLS.—Within a flowchart, graphic symbols are used to specify arithmetic operations and relational conditions. The following are commonly-used arithmetic and relational symbols.

+	plus, add
−	minus, subtract
*	multiply
/	divide
±	plus or minus
=	equal to
>	greater than
<	less than
≥	greater than or equal to
≤	less than or equal to
≠	not equal to
⋡	not greater than
⋠	not less than
YES or Y	Yes
NO or N	No
TRUE or T	True
FALSE or F	False

FLOWCHARTING TEMPLATE.—To aid in drawing the flowcharting symbols, you may use a flowcharting template. Figure 1-3 shows a template containing the standard symbol cutouts. A template is usually made of plastic with the symbols cut out to allow tracing the outline.

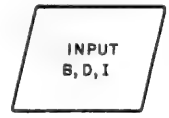
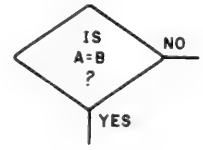
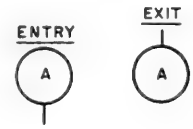
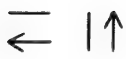
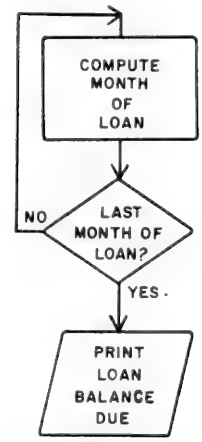
SYMBOL	DEFINITION	EXAMPLE	EXPLANATION
	**PROCESS SYMBOL** is used to represent general processing functions not represented by other symbols. It depicts the process of operations resulting in a change of value, form, or location of information.		Divide I by 12 assign value to R.
	**INPUT/OUTPUT SYMBOL** is used to represent any function of an I/O device. Making information available for processing is an Input function; recording processed information is an Output function.		Enter these values through the terminal, store in locations B, D, I.
	**DECISION SYMBOL** is used to depict a point in a program at which a branch to one of two or more alternate paths is possible.		If A is NOT equal to B, take NO branch. If A is equal to B, take YES branch.
	**TERMINAL, INTERRUPT SYMBOL** represents a terminal point in a flowchart, for example, start, stop, halt, delay, or interrupt.		START/STOP flow chart at this point.
	**CONNECTOR SYMBOL** represents a junction in a line of flow to another part of the flowchart. A common identifier, such as an alphabetic character, number, or mnemonic label, is placed within the exit and its associated entry.		This represents the EXIT point and the ENTRY point in a flowchart.
	**FLOWLINE SYMBOL** is used to represent flow direction by lines drawn between symbols. Normal direction of flow is left to right and top to bottom. If the direction of flow is other than normal, arrowheads are required at the point of entry.		Initial processing is shown here. If the NO branch is taken, the processing block is performed again. If the YES branch is taken, the INPUT/OUTPUT operation is performed.

Figure 1-2.—Fundamental Flowcharting symbols.

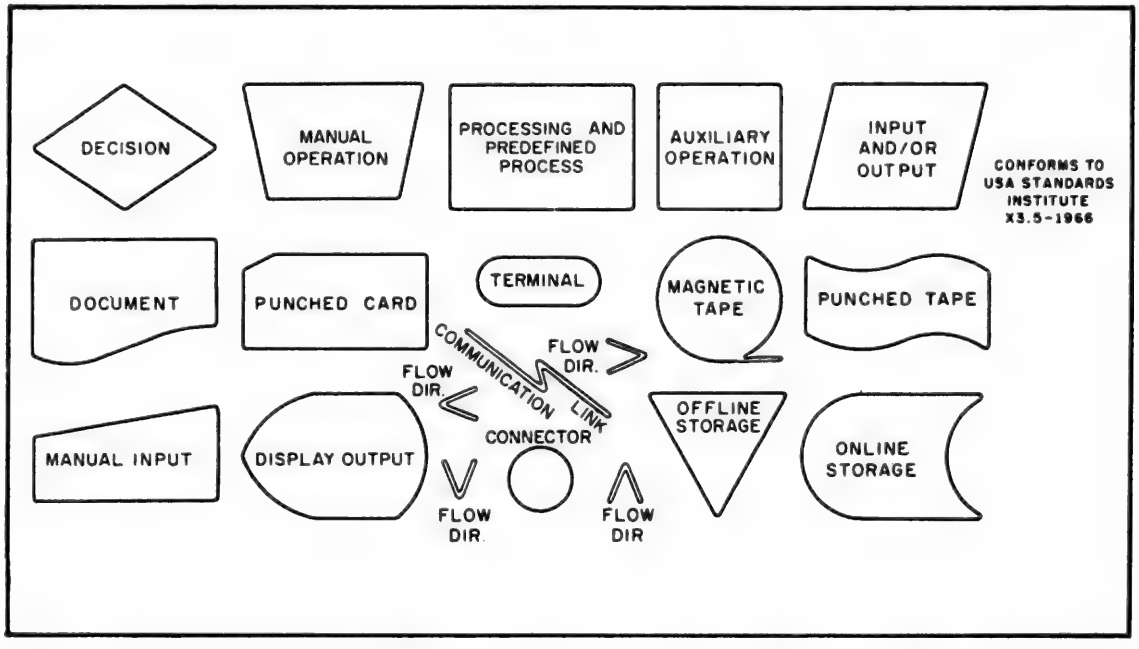


Figure 1-3.—Flowchart Template.

FLOWCHART WORKSHEET.—The Flowchart Worksheet is a means of standardizing documentation. It provides space for drawing programming flowcharts and contains an area for identification of the job, including application, procedure, date and page numbers (fig. 1-4). You may find it helpful when you develop flowcharts. If you don't have this form available, a plain piece of paper will do.

Constructing a Flowchart

There is no "best way" to construct a flowchart. There is no way to standardize problem solution. Flowcharting and programming techniques are often unique and conform to the individual's own methods or direction of problem solution.

This manual will show an example of developing a programming flowchart. It is not the intent to say this is the best way; rather, it is one way to do it.

By following this text example you should grasp the idea of solving problems through flowchart construction. As you gain experience and familiarity with a computer system, these ideas will serve as a foundation.

In order to develop a flowchart, you must first know what problem you are to solve. It is then your job to study the problem definition and develop a flowchart to show the logic, steps, and sequence of steps the computer is to execute in order to solve the problem.

As an example, suppose you have taken a short-term second mortgage on a new home, and you want to determine what your real costs will be: the amount of interest; the amount to be applied to principal; and the final payment at the end of the three year loan period.

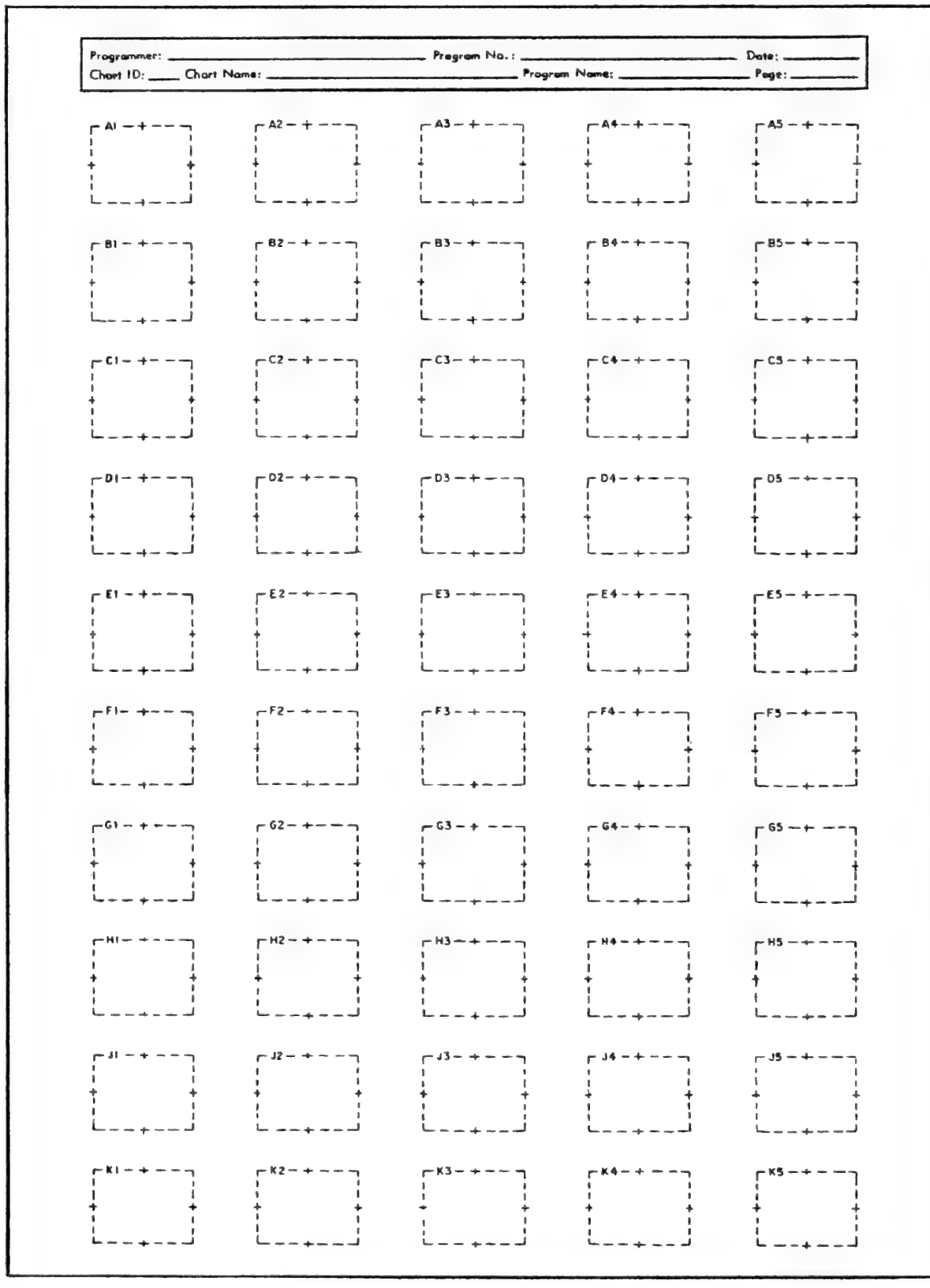


Figure 1-4.—Flowchart Worksheet.

The first step is to be sure you understand the problem completely—What are the inputs and the outputs and what steps are needed to answer the questions? Even when you are specifying a problem of your own, you'll find we don't usually think in small detailed sequential steps. But, that is exactly how a computer operates; one step after another in a specified order. Therefore, it is necessary for you to think the problem solution through step-by-step. You might clarify the problem as shown by the Problem Definition in figure 1-5.

After you have this level of narrative problem definition, you are ready to develop a flowchart showing the logic, steps, and sequence of steps you want the computer to execute in order to solve the problem. A programming flowchart of this problem is also shown in figure 1-5.

You now have a plan of what you want the computer to do. The next step is to code a program that can be translated by a computer into a set of instructions it can execute. This step is called program coding.

PROGRAM CODING

It is important to remember program coding is not the first step of programming. Too often we have a tendency to start coding too soon. As we discussed earlier, there is a great deal of planning and preparation to be done prior to sitting down to code the computer instructions to solve a problem. For the example amortization problem (fig. 1-5), we have analyzed the specifications in terms of (1) the output desired; (2) the operations and procedures required to produce the output; and (3) the input data needed. In conjunction with this analysis, we have developed a programming flowchart which outlines the procedures for taking the input data and processing it into usable output. You are now ready to code the instructions that will control the computer during processing. This requires that you know a programming language.

Before getting into the specific programming language called BASIC, it may be helpful to have a greater understanding of programming languages in general.

All programming languages are composed of instructions that enable the computer to process a particular application, or perform a particular function.

Instructions

The instruction is the fundamental element in program preparation. Like a sentence, an instruction consists of a subject and a predicate. However, the subject is usually not specifically mentioned; rather it is some implied part of the computer system directed to execute the command that is given. For example, the chief tells a sailor to "dump the trash." The sailor will interpret this instruction correctly even though the subject "you" is omitted. Similarly, if the computer is told to "ADD 1234," the control unit may interpret this to mean that the arithmetic-logic unit is to add the contents of address 1234 to the contents of the accumulator.

In addition to an implied subject, every computer instruction has an explicit predicate consisting of at least two parts. The first part is referred to as the command, or operation; it answers the question "what?." It tells the

PROBLEM DEFINITION

MORTGAGE AMORTIZATION—This program is to determine the monthly amount of interest (A) and amount applied to the principal (P) of the mortgage giving the balance (B) at the end of a thirty-six month period.

INPUT: The monthly payment is to be entered as variable D, the beginning balance of the mortgage is to be entered as variable B, and the annual interest rate is to be entered as variable I. This input is to be entered into the system via the terminal.

OUTPUT: The end result is to be a listing displaying the amount applied to principal and interest and the current loan balance each month, with one final entry showing the final payment on the mortgage.

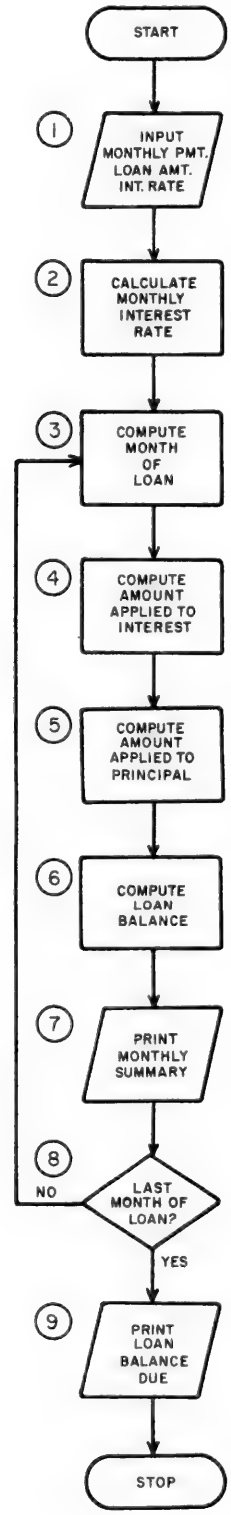


Figure 1-5.—Problem Definition and Programming Flowchart.

computer what operation it is to perform; i.e., read, print, input. Each machine has a limited number of built-in operations that it is capable of executing. An operation code is used to communicate the programmer's intent to the computer.

The second specific part of the predicate, known as the operand names the object of the operation. In general, the operand answers the question "where?." Operands may indicate the following:

1. The location where data to be processed is found.
2. The location where the result of processing is to be stored.
3. The location where the next instruction to be executed is found. (When this type of operand is not specified, the instructions are executed in sequence.)

The number of operands and the structure or format of the instructions vary from one computer to another. However, the operation always comes first in the instruction and is followed by the operand(s). The programmer must prepare instructions according to the format required by the language and the computer to be used.

Instruction Set

The number of instructions in a computer's instruction set may range from less than 30 to more than 100. These instructions may be classified into categories such as input/output (I/O), data movement, arithmetic, logic, and transfer of control. Input/output instructions are used to communicate between I/O devices and the central processor. Data movement instructions are used for copying data from one storage location to another and for rearranging and changing of data elements in some prescribed manner.

Arithmetic instructions permit addition, subtraction, multiplication, and division. They are common in all digital computers. Logic instructions allow comparison between variables, or between variables and constants. Transfer of control instructions are of two types, conditional or unconditional. Conditional transfer instructions are used to branch or change the sequence of program control, depending on the outcome of the comparison. If the outcome of a comparison is true, control is transferred to a specific statement number; if it proves false, processing continues sequentially through the program. Unconditional transfer instructions are used to change the sequence of program control to a specified program statement regardless of any condition.

Programming Languages

Programmers must use a language that can be understood by the computer. There are several methods that can achieve human-computer communication. For example, let us assume the computer only understands French and the programmer speaks English. The question arises: How are we to communicate with the computer? One approach is for the programmer to code the instructions with the help of a translating dictionary prior to giving them to the processor. This would be fine so far as the computer is concerned; however, it would be very awkward for the programmer.

Another approach is a compromise between the programmer and computer. The programmer first writes instructions in a code that is easier to relate to English. This code is not the computer's language; therefore, it does not understand the orders. The programmer solves this problem by giving the computer another program, one that enables it to translate the instruction code into its own language. This translation program, for example, would be equivalent to an English-to-French dictionary, leaving the translating job to be done by the computer.

The third and most desirable approach from an individual's standpoint, is for the computer to accept and interpret instructions written in everyday English terms. Each of these approaches has its place in the evolution of programming languages and is used in computers today. The first approach is known as machine language, the second as symbolic, and the third as procedure-oriented.

MACHINE LANGUAGES.—With early computers, the programmer had to translate instructions into the machine language form that the computers understood. This language was a string of numbers that represented the instruction code and operand address(es).

In addition to remembering dozens of code numbers for the instructions in the computer's instruction set, the programmer also had to keep track of the storage locations of data and instructions. This process was very time consuming, quite expensive and often resulted in errors. Correcting errors or making modifications to these programs was a very tedious process.

SYMBOLIC LANGUAGES.—In the early 1950s, *mnemonic* instruction codes and symbolic addresses were developed. This improved the program preparation process by substituting letter symbols (mnemonic codes) for basic machine language instruction codes. Each computer has mnemonic code, although the symbols vary among the different makes and models of computers. The computer still uses machine language in actual processing, but it translates the symbolic language into machine language equivalent. Symbolic languages have many advantages over machine language coding in that less time is required to write a program, detail is reduced, and fewer errors are made. Errors which are made are easier to find, and programs are easier to modify.

PROCEDURE-ORIENTED LANGUAGES.—The development of mnemonic techniques and *macroinstructions* led to the development of procedure-oriented languages. These languages are oriented toward a specific class of processing problems. A class of similar problems is isolated, and a language is developed to process these types of applications. Several languages have been designed to process problems of a scientific-mathematical nature and others that emphasize file processing. The most familiar of these are BASIC and FORTRAN for scientific or mathematical problems, and COBOL for file processing.

Programs written in procedure-oriented languages, unlike those in symbolic languages, may be used with a number of different computer makes and models. This feature greatly reduces reprogramming expenses when changing from one computer system to another. Other advantages to procedure-oriented languages are: (1) they are easier to learn than symbolic

languages; (2) they require less time to write; (3) they provide better documentation; and (4) they are easier to maintain. However, there are some disadvantages of procedure-oriented languages. They require more space in memory and they process data at a slower rate than symbolic languages.

Coding a Program

Regardless of the language used, there are strict rules the programmer must adhere to with regard to punctuation and statement structure when coding any program. Using the programming flowchart introduced earlier, we have now added a program coded in BASIC to show the relationship of the flowchart to the actual coded instructions (fig. 1-6). Don't worry about complete understanding, just look at the instructions with the flowchart to get an idea of what coded instructions look like.

You will have to have specific information about the computer you are to use and how the language is implemented on that particular computer. The computer manufacturers provide these specifics in their user's manual. **Get a copy and study it before you begin to code.** The differences may seem minor to you, but they may prevent your program from running.

Once coding is completed, the program must be debugged and tested prior to implementation.

Debugging

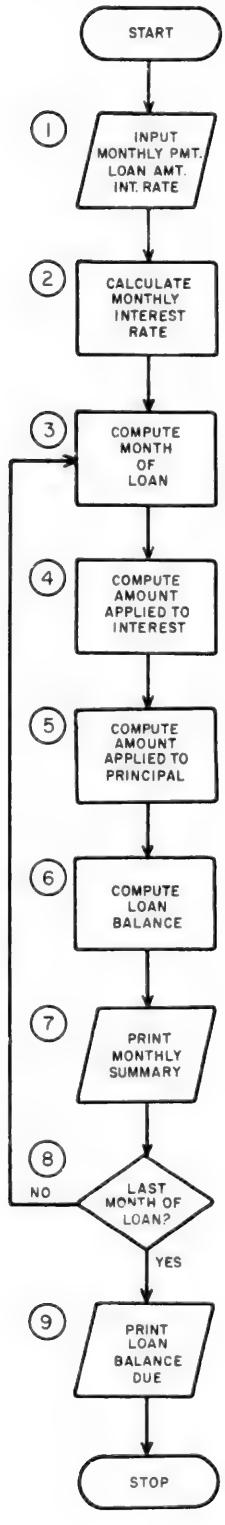
Errors caused by faulty logic and coding mistakes are referred to as "bugs." Finding and correcting these mistakes and errors that prevent the program from running and producing correct output is called "debugging."

Rarely do complex programs run to completion on the first attempt. Often, time spent debugging and testing equals or exceeds the time spent in program coding. This is particularly true if insufficient time was spent on problem definition and logic development. Some common mistakes which cause program bugs are: mistakes in coding punctuation, incorrect operation codes, transposed characters, keying errors and failure to provide a sequence of instructions (a program path) needed to process certain conditions.

To reduce the number of errors, you will want to carefully check the coding sheets before they are turned in for keying. This process is known as "desk-checking" and should include an examination for program completeness. Typical input data should be manually traced through the program processing paths to identify possible errors. In effect, you will be attempting to play the role of the computer. After the program has been desk-checked for accuracy, the program is ready to be *assembled* or *compiled.* Assembly and compiler programs prepare your program (source program) to be executed by the computer and they have error diagnostic features which detect certain types of mistakes in your program. These mistakes must be corrected. Even when an error-free pass of the program through the assembly or compiler program is accomplished, this does not mean your program is perfected. However, it usually means the program is ready for testing.

Testing

Once a program reaches the testing stage, generally, it has proven it will run and produce output. The purpose of testing is to determine that all




```
10 PRINT "ENTER MONTHLY PAYMENT, ";
20 PRINT "LOAN AMOUNT, INTEREST RATE"
30 INPUT D,B,I
40 LET R=I/12
50 FOR M=1 TO 36
60 LET A=B*R
70 LET P=D-A
80 LET B=B-P
90 PRINT M;D;B,P,A
100 NEXT M
110 PRINT "WITH ONE FINAL PAYMENT OF",B
120 END
```

Figure 1-6.—Programming Flowchart and Coded Program.

data can be processed correctly and that the output is correct. The testing process involves processing input test data that will produce known results. The test data should include: (1) typical data, which will test the commonly used program paths; (2) unusual but valid data, which will test the program paths used to process exceptions; and (3) incorrect, incomplete, or inappropriate data, which will test the program's error routines. If the program does not pass these tests, more testing is required. You will have to examine the errors and review the coding to make the coding corrections needed. When the program passes these tests, it is ready for computer implementation. Before computer implementation takes place, documentation must be completed.

Documentation

Documentation is a continuous process, beginning with the problem definition. Documentation involves collecting, organizing, storing, and otherwise maintaining a complete record of the programs and other documents associated with the data processing system.

The Navy has established documentation standards to ensure completeness and uniformity for computer system information between commands and between civilian and Navy organizations. SECNAVINST 5233.1 (Series) establishes minimum documentation requirements.

Local minimum documentation requirements are usually established by the head of the data processing department/division. At most commands this function is delegated to the project manager. The key to the minimum amount of documentation required by local commands should be the amount that is required for replacement personnel to understand input, processing, and output for each program or system for which they will be responsible.

A documentation package should include:

1. A definition of the problem. Why was the program written? What were the objectives? Who requested the program, and who approved it? These are the types of questions that should be answered.
2. A description of the system. The system environment (hardware, software, and organization) in which the program functions should be described (including systems flowcharts). General systems specifications outlining the scope of the problem, the form and type of input data to be used, and the form and type of output required should be clearly defined.
3. A description of the program. Programming flowcharts, program listings, program controls, test data and test results, storage dumps—these and other documents that describe the program and give a historical record of problems and/or changes should be included.
4. Operator instructions. Items that should be included are computer switch settings, loading and unloading procedures, and starting, running, and termination procedures.

Implementation

After the documentation has been completed, and the user has reviewed and accepted the test output, the project request is submitted to upper management, usually the ADP department head, for production approval. Once

upper management has approved the program, it can be put into production. If a program is to replace a program in an existing system, it is generally wise to have a period of *parallel processing*; that is, the job application is processed both by the old program and by the new program. The purpose of this period is to verify processing accuracy and completeness.

Once the program is in production it may be necessary to make modifications to the program to satisfy changing requirements. This is another important duty of the programmer, and it is not unusual to find programmers spending 25 percent of their time on this program maintenance activity. In some installations, there are programmers who do nothing but maintain production programs.

SUMMARY

The first step in the solution of any problem involves a fundamental but often overlooked concept—a thorough understanding of the problem. The second step in successful problem solving involves creating a flowchart showing the steps required to solve the problem.

Flowcharting is a pictorial means of representing a procedural solution to a problem in which different shaped symbols are used to represent operations, data, flow, equipment and so forth. There are two types of flowcharts—system (data) and programming. The tools of flowcharting are: (1) fundamental symbols; (2) graphic symbols; (3) flowcharting template; and (4) flowcharting worksheet.

The problem definition and flowchart development steps must be done prior to sitting down to code the computer instructions to solve a problem. Regardless of the language used, there are strict rules you must adhere to with regard to punctuation and statement structure when coding a program.

Once the program is coded, there are several phases that must be done before it can be put into production. These are desk-checking, debugging, testing, documentation and finally, implementation.

CHAPTER 1

EXERCISES

1. Arrange the following problem-solving steps in the correct sequence.

_____ Code the program, prepare the code in machine readable form, prepare test data, and perform debug and test runs.

_____ Develop the program logic and prepare a programming flowchart or decision table.

_____ Complete the documentation and prepare operator procedures for implementation and production.

_____ State, analyze, and define the problem.

2. Draw the flowcharting symbol that represents reading input data.

3. Refer to Figure 1-6. What block in the programming flowchart is equivalent to line 110 of the coded program?

4. Draw the flowcharting symbol that is used to depict the evaluation of a YES/NO, or TRUE/FALSE condition.

5. PROBLEM DEFINITION

   SALES COMMISSION—This program is to determine the amount of commission a salesperson is due.

   INPUT: Amount of sales will be input as S and percent commission will be input as P.

   OUTPUT: The end result will be a listing, displaying amount of sales and amount of commission.

   Using this problem definition, draw a programming flowchart depicting the programming steps required to solve the problem.

CHAPTER 1

EXERCISE ANSWERS

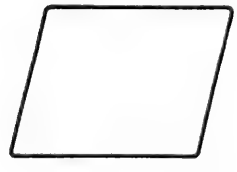
1. Problem-solving steps

   Step 1. State, analyze, and define the problem.

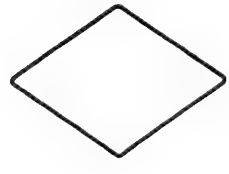
   Step 2. Develop the program logic and prepare a programming flowchart or decision table.

   Step 3. Code the program, prepare the code in machine readable form, prepare test data, and perform debug and test runs.

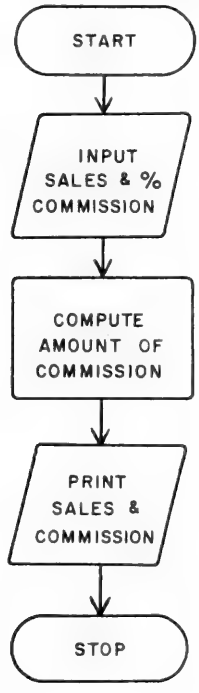
   Step 4. Complete the documentation and prepare operator procedures for implementation and production.

2.  INPUT/OUTPUT SYMBOL

3. 9

4.  DECISION SYMBOL

5. Sales Commission Flowchart



CHAPTER 2

INTRODUCTION TO BASIC

Fundamental Concepts and Language Structure

BASIC is different from other programming languages in its concern for the inexperienced user. Although it is a general-purpose programming language, it is designed primarily to be easy to learn, easy to use, and easy to remember. BASIC is oriented toward, but not restricted to, interactive use. Its statement structure is kept simple, and special rules are kept to a minimum. A BASIC program is meant to be simple so that even a novice is able to determine what the program is expected to do on the basis of examination. Only a little knowledge of BASIC is required to solve simple problems.

Simple BASIC programs can be used much like a small calculator. You might want to ask the computer to multiply 4 times 25 and print the results. The following example shows you how to accomplish this:

```
10  PRINT 4*25
20  END
RUN
 100
```

You keyed three lines into the computer. In the first line, line number 10, you told the computer to calculate 4 times 25 and print the answer (* in BASIC means multiply). In the second line, line number 20, you told the computer there were no more instructions in the program. The last command you gave the computer was **RUN,** a system command which tells the computer to execute the instructions and give the results. The computer multiplied 4 times 25 and gave the answer, which is 100.

As simple as it may seem, this is a computer program. The same function which took only one statement in BASIC, would take several lines of coding in other high-level programming languages such as COBOL or FORTRAN.

STATEMENT STRUCTURE

The instructions which are preceded by line numbers are called statements. A complete set of statements to solve a problem is called a program. The very last statement in each program must be the **END** statement.

Program statements in the BASIC language can be constructed in free form. The various parts must all appear, however, and be given in a definite order, as shown below:

10	**LET**	**A = 5**
statement number	keyword	descriptive information to support the keyword

STATEMENT NUMBER.—(Frequently referred to as line number). This number has two vital functions: (1) to identify the statement itself (statement label); and (2) to indicate to the BASIC compiler (interpreter) where you want this statement placed in the program sequence. The statement number must be an integer (a whole number—no decimal parts or fractions). For the purpose of this text, the range for statement numbers is from 1 to 99999. However, the number of digits may vary depending on the computer you are using. Table 2-1 shows examples of statements with valid and invalid statement numbers.

Table 2-1.—BASIC Statement Numbers

Statements with valid statement numbers	Statements with invalid statement numbers	Explanation
10 LET A = 5	**−10 LET A= 5**	Cannot contain minus sign
40 INPUT I,Y,L	**+99 INPUT I,Y,L**	Cannot contain plus sign
99 PRINT I,Y,L	**5.99 PRINT I,Y,L**	Cannot contain decimal point
99999 END	**999999 END**	Too large

BASIC LANGUAGE KEYWORD.—Keywords are used to tell the computer what function is to be performed by this statement. For example, LET, PRINT, INPUT, and END as shown in Table 2-1. These and other keywords will be introduced and discussed as appropriate throughout the remaining chapters. The spelling of each keyword must be exact, or the compiler (interpreter) will tell you this is an invalid statement.

DESCRIPTIVE INFORMATION.—This information completes the description of the function to be performed and varies with the keyword used. See the first three examples in Table 2-1. There are a few instances where no additional information is required or allowed. See the last example in Table 2-1.

In order to write these statements, you must first know the syntax; that is, the characters and symbols used to construct the statements, as well as the rules, conventions, and special features of the language. This includes the character set and the methods used to represent numbers and predefined functions.

The BASIC Character Set

There are three types of characters used in BASIC. These are: (1) alphabetic, (2) numeric, and (3) special characters.

ALPHABETIC CHARACTERS.—The alphabetic characters used in BASIC are the standard English alphabet, A through Z.

NUMERIC CHARACTERS.—The numeric characters used in BASIC are the digits 0 through 9.

SPECIAL CHARACTERS.—The following are special characters used in BASIC:

	Blank	'	Single quotation mark
=	Equal sign or assignment symbol	"	Double quotation mark
+	Plus sign	;	Semicolon
–	Minus sign	:	Colon
*	Asterisk or multiply symbol	!	Exclamation symbol
/	Slash or divide symbol	?	Question mark
↑	Up-arrow or exponentiation symbol	&	Ampersand
		<	Less than symbol
)	Right parenthesis	>	Greater than symbol
(	Left parenthesis	#	Number or pound sign
,	Comma	$	Dollar sign
.	Point or period	%	Percent sign

OTHER SPECIAL CHARACTERS.—Some special characters are combined to form other elements in BASIC. The following list shows these combinations:

> = greater than or equal

< = less than or equal

< > not equal

** exponentiation

BASIC Numbers

When you are using numeric data in a BASIC program, there are certain conventions that must be adhered to. You cannot use "$" (dollar sign), ","

(comma), or the "/" (slash) in a BASIC number. There are also restrictions on the number of digits that can be used in one data element. The number of digits may vary depending upon the computer you are using. Refer to your user's manual for specific instructions. Table 2-2 shows examples of correctly and incorrectly coded BASIC numbers.

Table 2-2.—BASIC Numbers

Valid BASIC Numbers	Invalid BASIC Numbers	Explanation
99.95	$99.95	Dollar sign cannot be used
100000.00	100,000.00	Comma cannot be used
+5678901.2	+567890100	Too many digits, only eight are allowed (decimal points, positive and negative signs do not count).
.25	1/4	Slash cannot be used; however, this is a valid expression as an arithmetic operation (1 divided by 4).

SCIENTIFIC NOTATION.—As seen in Table 2-2, in the third example, we have a number, +567890100, with too many digits. You may ask, how do we represent very large and very small numbers? Scientific notation is used. In scientific notation numbers are expressed in terms of a figure between 1 and 10 times a power of 10. The number 567890100 would be written 5.678901×10^8 in scientific notation. This method uses exponent form and is commonly called E notation, E form or E format. In BASIC, E notation is formed by adding the letter E and a positive or negative integer to a BASIC number. For example, E8 means "times 10 to the 8th power."

$$5.678901E8 = 5.678901 \times 10^8 = 567890100$$

Table 2-3 shows other examples of the use of E notation.

Table 2-3.—E Notation

Number using E notation	Interpreted as	Explanation
5.5E8	550,000,000	The decimal has been moved eight places to the right
5.5E-8	0.000000055	The decimal has been moved eight places to the left
−2.54321E10	−25,432,100,000	The decimal has been moved ten places to the right
−2.54321E-5	−0.0000254321	The decimal has been moved five places to the left

Since we're multiplying by powers of 10, the integer following the E indicates how many positions and in what direction to move the decimal point. If the integer after the letter "E" is negative, you move the decimal point to the left; if the integer is positive, you move the decimal point to the right.

Predefined Functions

Some mathematical functions, for example, square root and tangent, are used so frequently that they have been incorporated into the BASIC language as predefined functions. You can use these to compute the same mathematical function with many different values. Rather than writing the coding required to do each calculation, you may use the "function" capability of the BASIC language. For example:

```
LET X = SQR (36)
```

This will cause the computer to calculate the square root of 36 and assign that value to X. Table 2-4 shows a list of some of these predefined mathematical functions.

Table 2-4.—Predefined Functions

FUNCTION	DESCRIPTION
ABS (X)	Absolute value of X
ATN (X)	Arc tangent of X in radian measure
COS (X)	Cosine of X in radian measure
EXP (X)	Natural exponential of X
INT (X)	The largest integer not greater than X
LOG (X)	Natural logarithm of X (base e)
RND (X)	Generates random numbers between 0 and 1
SGN (X)	Algebraic "sign" of X: -1 if $X<0$, 0 if $X=0$, and $+1$ if $X>0$
SIN (X)	Sine of X in radian measure
SQR (X)	Square root of X
TAN (X)	Tangent of X in radian measure

FUNDAMENTAL CONCEPTS

Up to this point, we have been discussing the mechanics of the BASIC programming language. We will now discuss fundamental concepts and how

you use these in your interaction with the computer in keying in and running your programs. This will include more about statement numbers, spacing within statements, keying statements into the computer, methods of correcting mistakes, how the computer responds when your program contains a syntax error, and the use of the **REMARK** statement.

Assigning Statement Numbers

Duplicate statement numbers are not allowed. If two statements are entered with the same statement number, the computer will accept the second statement with the duplicate statement number and replace the first.

When assigning statement numbers, it is a good idea to increment them by 10; this will allow you to insert additional statements between existing statements in your program later. This is not mandatory, but it is a good practice. This technique will prevent you from having to completely renumber a program, if you find you need to add a statement after you have completed writing your program. The following example shows how this technique works:

```
10   LET A = 2
20   LET B = 4
30   LET C = 6
40   LET X = (A+C)/B
50   LET Y = (A*C)/B
60   LET Z = A+B+C
70   END
65   PRINT X,Y,Z
```

In this example the PRINT statement, which is needed to print the results from your calculations was omitted. Since the line numbers were incremented by 10, it was easy to assign the PRINT statement a line number between 60 and 70.

When the program is run or you list your program, the PRINT statement, line number 65, will be inserted in its proper place by the computer.

```
10   LET A = 2
20   LET B = 4
30   LET C = 6
40   LET X = (A+C)/B
50   LET Y = (A*C)/B
60   LET Z = A+B+C
65   PRINT X,Y,Z
70   END
RUN
 2             3             12
```

Spacing Within Statements

Spacing within statements in your program is not very important, the computer generally ignores spaces except those within quotation marks. For example, one of the sample programs used earlier could be written this way:

```
10PRINT4*25
20END
```

OR:

```
10   PRINT    4    *   25
20   END
```

or various other ways. However, as a matter of practice, you will want to use spacing that provides clarity and maintains the integrity of the statement. Appropriate use of spaces within a line will make the line easier for you and others to read and understand.

The sample program below is an example of the recommended spacing within statements:

```
10   REMARK THIS PROGRAM CONVERTS INCHES TO CENTIMETERS
20   INPUT I
30   LET C = I*2.54
40   PRINT "INCHES","CENTIMETERS"
50   PRINT I,C
60   END
```

In this example, we have used spacing techniques similar to those you would use in ordinary typing.

KEYING IN A PROGRAM

Once you have written a program and have a computer available, you are ready to key in your BASIC program. The procedures for this will vary, depending upon the computer system you are using. Consult your user's manual for procedures for your specific computer.

Keying Statements

To enter BASIC statements, you key in the statement beginning with the statement (line) number, after the prompt on the terminal. Once you have completed the statement, depress the **RETURN** key. Only one statement may be entered per line. Some computers are equipped with an automatic line numbering feature which will automatically number your statements, incrementing them by 10.

It is not necessary to key in your BASIC statements in sequence by line number. The computer will sort the statements by line number and place them in ascending sequence, regardless of the sequence in which you keyed them in. The following example shows what we mean:

```
10   REMARK THIS PROGRAM CONVERTS INCHES TO CENTIMETERS
20   INPUT I
30   LET C = I*2.54
40   PRINT "INCHES","CENTIMETERS"
60   END
50   PRINT I,C
LIST
```

You forgot to type in line 50 when you initially entered your program, but that is not a catastrophe. You can enter it after line 60. By entering the system command **LIST**, you will find line 50 has been placed in the proper sequence in the program. Your listing will look like this:

```
10   REMARK THIS PROGRAM CONVERTS INCHES TO CENTIMETERS
20   INPUT I
30   LET C = I*2.54
40   PRINT "INCHES","CENTIMETERS"
50   PRINT I,C
60   END
```

Correcting Mistakes

At one time or another, we all make mistakes when we're keying in statements. For that reason, ways are provided for you to make corrections. These may vary depending on the computer you're using. On many computers, the statements you key in are not sent to the interpreter until you have depressed the RETURN key. Should you make a mistake in a statement before you have depressed the return key, you can backspace with the backspace key and key the correct information.

For example:

you typed

```
10   LET A = B+
```

but you intended to type an asterisk (multiply) instead of a plus sign (addition). You can correct it by backspacing once and keying the correct information. The * (asterisk) will replace the + (plus), then you continue by keying the rest of the statement.

Should you make more than one mistake in a line, use one backspace for each character you want to erase or replace.

If you make a mistake in a statement and you have already entered the statement, you will have to correct it in another way. The following examples show how it can be done.

Suppose you had entered the following program:

```
10  LTE A=2
20  LET B=4
30  LET C=6
40  LET X=(A+C)/B
50  LET Y=(A*C)/B
60  LET Z=A+B+C
70  PRINR X,Y,Z
80  END
```

Lines 10 and 70 contain errors. You can correct them by simply retyping the lines as they should be.

```
10  LET A=2
70  PRINT X,Y,Z
```

These will overlay (replace) the original lines 10 and 70. If you list or display your program after making these changes, it will appear as follows:

```
10  LET A=2
20  LET B=4
30  LET C=6
40  LET X=(A+C)/B
50  LET Y=(A*C)/B
60  LET Z=A+B+C
70  PRINT X,Y,Z
80  END
```

Lines 10 and 70 which were originally entered incorrectly were replaced by the retyped lines and are in correct form and in proper sequence in the

program. As we stated earlier, duplicate line numbers are not allowed and the second statement entered with the same line number will overlay or replace the first.

Some computers have an EDIT/RECALL feature which allows you to enter the line number you want to correct and then depress **EDIT/RECALL**. It will display this line on the screen and allow you to make changes to it.

You may also delete a line by typing the command **DELETE** and the line number, for example:

```
DELETE   50
```

This will delete line 50.

Another way to delete a line is to type in only the line number and depress the RETURN key, for example:

```
50
```

This will also delete line 50.

When you are keying in your program, some computers will check for syntax errors as each statement is entered. Others will check for syntax errors when you enter the system command RUN. In either case, you'll get an error message indicating where there is a syntax error. (See the following example.) These errors must be corrected before the program can be executed.

```
10   LET W=2
20   LET X=4
30   LET Y=6
40   LET Z=8
50   PRINT W+6,X*Y,Z-X,W/X
60   PRINT W,X,)Y,Z
70   END
RUN
LINE 60 SYNTAX ERROR IN EXPRESSION
```

Look at line 60, find the error. The **)Y** is incorrect; the **)** should not be there. Now that you know what the error is, you can use one of the methods discussed to correct it and try again.

Unfortunately, at this stage, the computer can only find syntax errors, it cannot determine if you have made a logic error. Logic errors will be found when you try to run the program during debugging and testing.

To help find logic errors, you may find it useful to include some documentation about the program within it. To do this, you use the REMARK statement.

REMARK Statement

All languages provide the capability for inserting programmer comments to make the program listing more readable and to aid in testing and documenting the program. In BASIC, the keyword **REMARK** (which may be abbreviated **REM**) is used. The REMARK statement is a nonexecutable statement; that is, the REM and following comments appear only in the listing of the program and do not, in any way, affect execution of the program.

FORMAT:

```
10    REMARK    THIS PROGRAM CONVERTS INCHES TO CENTIMETERS
```

statement number	keyword	any programmer comments

When a statement with the keyword REMARK is used, the entire line on which the statement appears is considered a comment.

The following example shows how the REMARK statement is used to document a program so that anyone can understand what the program is used for:

```
10  REMARK THIS PROGRAM IS USED TO CONVERT INCHES
20  REMARK TO CENTIMETERS. WHEN THE SYSTEM
30  REMARK ASKS FOR INPUT, INPUT THE NUMBER OF
40  REMARK INCHES YOU WANT CONVERTED TO CENTIMETERS
50  INPUT I
60  LET C = I*2.54
70  PRINT "INCHES","CENTIMETERS"
80  PRINT I,C
90  END
```

Not only can REMARK statements be used at the beginning of a program, they can be used throughout the program to separate different segments of a program and to introduce subroutines or loops.

SUMMARY

BASIC is designed to be easy to learn, easy to use, and easy to remember. BASIC programs are meant to be simple so that even a novice can understand them.

Instructions which are preceded by line numbers are called program statements. The parts of a statement are: *statement (line) number, keyword, and descriptive information.*

The BASIC character set is divided into three categories: alphabetic, numeric, and special characters. E notation (scientific notation) is used for representing very large and very small numbers. BASIC has included several predefined functions so that you don't have to code some of the more common mathematical functions.

Line numbers tell the computer the sequence of the instructions in your program. It is a good practice to increment your line numbers by 10 to allow for inserting additional statements between existing statements in your program. Spacing within a line is not important; however, appropriate use of spaces within a line makes it easier for you and others to read and understand. The first step in entering in a statement is keying in the line number.

You do not have to key the BASIC statements in sequence, the computer sorts them into line number sequence. The system command **LIST** causes your program to be listed with the statements in ascending sequence by line number. The system command **RUN** is used to tell the computer to execute the program.

Unfortunately the computer does not detect logic errors; you will have to find those through testing and debugging.

The **REMARK** statement is used to aid in documenting your program.

CHAPTER 2

EXERCISES

1. Arrange the following statement elements in proper order according to the BASIC statement structure and label the parts.

A,B,C 40 PRINT

2. Which of the following are correctly coded BASIC line numbers? For those incorrectly coded, what is wrong with them?

A. 20
B. 25
C. 30
D. 1/4
E. 9.99
F. 54321
G. 123456
H. 99999
I. #9999
J. .2500

3. Convert the following numbers in E notation into the numbers they represent.

A. 2.5E3
B. −1.5E2
C. 9.9E-5
D. −3.5E6

4. Which of the following are correctly coded numbers? For those incorrectly coded, what is wrong with them?

A. 95
B. 9,500
C. $100.00
D. 1/4
E. .95
F. 52
G. 65$
H. 74

5. A. If you typed the following BASIC program into the computer and gave the system command LIST, what would the listing look like?

```
10  LET A=2
20  LET C=6
30  PRINT A+B+C
20  LET C=8
15  LET B=4
40  END
25  PRINT "TOTAL"
```

B. If you then typed the system command RUN, what would be printed?

CHAPTER 2

EXERCISE ANSWERS

1. 40 PRINT A,B,C,

   line #, keyword, descriptive information

2. correctly coded line numbers

   A,B,C,F,H

   incorrectly coded line numbers

   D. 1/4 must be a whole number
   E. 9.99 cannot contain a decimal point
   G. 123456 maximum number allowed is 99999
   I. #9999 cannot contain special characters
   J. .2500 cannot contain a decimal point

3. A. 2500.0
   B. −150.0
   C. 0.000099
   D. −3500000.0

4. correctly coded numbers

   A,E,F,H

   incorrectly coded numbers

   B. 9,500 cannot contain commas
   C. $100.00 cannot contain $
   D. 1/4 cannot contain special characters except decimal points
   G. 65$ cannot contain $

5. A.
```
10  LET A=2
15  LET B=4
20  LET C=8
25  PRINT "TOTAL"
30  PRINT A+B+C
40  END
```

   B. TOTAL
   14

CHAPTER 3

SOLVING SIMPLE PROBLEMS WITH BASIC

END, PRINT, LET, Constants, Variables, and Arithmetic Operations

WRITING SIMPLE BASIC PROGRAMS

Learning to solve simple problems in BASIC is somewhat like learning to write and communicate in a foreign language, with one exception—BASIC is much easier to learn. By comparison, in the average pocket dictionary there are approximately 50,000 words, but there are less than 50 keywords in BASIC.

You could probably learn all these keywords in one day; however, it takes longer than one day to become a proficient programmer in the BASIC language. As with any language, knowing only the grammar or mechanics of the language is not sufficient. You also must know how to use the language to instruct the computer to solve problems.

As discussed in Chapter 1, the first step in writing a program is problem definition and understanding. Whether it is an elaborate application or just a simple "one time" program you wish to write, you first must understand the problem, then decide how you are going to solve it. Once this is done, you start writing the program. Challenging as it is, this can be fun; although at times it can be very frustrating. As you progress through the course, make note of how each keyword is used to instruct the computer to produce the desired result.

In this chapter, we will discuss three statements that can be used in writing simple BASIC programs and give examples of programs. The three keywords used to construct these statements are END, PRINT, and LET.

END Statement

Every program requires an **END** statement, so we'll start with it and its functions. The END statement has two functions—to indicate to the compiler (interpreter) that there are no more BASIC statements for it to translate and to terminate execution of the program. Execution of an END statement causes the computer to print a message which indicates program execution is terminated and that the computer is ready for further processing.

The format of the END statement is:

```
999 END
```

statement number — keyword

Remember, the last statement in any BASIC program must be the single keyword END. This means that the statement number must be the largest statement number in the program. There is no additional information required or allowed in the END statement.

PRINT Statement

The next area of consideration is printing and the uses of the **PRINT** statement. BASIC is different from other high level programming languages in that it has a predefined format for printed output. This format, referred to as *standard* spacing, divides the print line into a specified number of print zones or fields of a predefined length. The number and length of print zones may vary with different computers. For the purpose of this text, we will use 16 spaces per print field. Additionally, *packed* spacing may be achieved through the use of punctuation in the PRINT statement.

The function of the PRINT statement is to instruct the computer to output something, either on the terminal or the printer. It can be used in several different ways. One way is to print the value of a single variable.

Example:

Printing the value of a single variable

```
30 PRINT A
99 END
RUN
```

This statement will cause the computer to print whatever value it has for the variable A. The output will be the value of A only, the variable name A will not be printed. Let's assume the value of A in the PRINT statement is 896. When the PRINT statement is executed, the output will be:

```
 896
```

Note that only the value of variable A was printed, not the variable name.

Now let's suppose you wanted to print the values of several variables. It could be done this way:

Example:

Printing the values of three variables

```
10 PRINT A,B,C
99 END
RUN
```

Assume the values of A, B, and C are 896, 754, and 969 respectively. The output from the PRINT statement would look like this:

```
 896          754          969
```

Whenever you wish to print several variables with one PRINT statement, you must separate them with either commas or semicolons, as was done in line number 10 of the example. The commas will give you standard spacing. If semicolons had been used, we would have gotten packed spacing. More on the use of the comma and semicolon will be discussed later in the chapter.

Printing the results of a computation is another use of the PRINT statement. Let's suppose we want to get the average of three scores: 82, 95, and 93. First we must define the problem: how to compute an average. You add up the scores and divide by the number of scores. This problem can be solved by using a PRINT statement like this:

Example:

Print the results of a computation

```
15 PRINT (82+95+93)/3

99 END

RUN
```

We have used a simple mathematical formula in conjunction with the PRINT statement to solve the problem. First the computer will add all the scores, then it will divide by three and print the result. When this statement is executed, the output will be:

```
 90
```

This output is rather plain and meaningless to anyone other than the person who wrote the program. There is another use of the PRINT statement which will enhance the output from the previous example, printing a message using the PRINT statement. Any message to be printed must be enclosed in quotation marks.

Example:

Printing a message

```
10 PRINT "THE AVERAGE SCORE IS"

20 PRINT (82+95+93)/3

30 END

RUN
```

The output from these PRINT statements would be:

```
THE AVERAGE SCORE IS
 90
```

You will notice the quotation marks around the message in line 10 causes the message to be printed just as it appears in the program. Using the PRINT statement in this manner adds clarity and meaning to the output.

Now let's look at another combination of uses of the PRINT statement. Suppose we wanted to print out the individual scores and the average of the scores, with a blank line between them. The following example shows how this could be done:

Example:

Printing blank lines

```
10 PRINT "INDIVIDUAL SCORES"
20 PRINT 82
30 PRINT 95
40 PRINT 93
50 PRINT
60 PRINT "THE AVERAGE SCORE IS"
70 PRINT (82+95+93)/3
80 END
RUN
```

Note that line 50 contains a blank PRINT statement. This will cause spacing between lines of output. It can be used to provide clarity and improve readability. When the program is executed, the output will look like this:

```
INDIVIDUAL SCORES
 82
 95
 93

THE AVERAGE SCORE IS
 90
```

As seen in this example the blank PRINT statement produced a blank line between the individual scores and the average score. This feature may work differently on some computers.

We could have written the previous example in a different manner—using semicolons to separate messages and numbers.

Example:

Printing messages and numbers on the same line

```
10 PRINT "INDIVIDUAL SCORES ARE ";82;95;93
20 PRINT
30 PRINT "THE AVERAGE SCORE IS ";(82+95+93)/3
40 END
RUN
```

The information enclosed in quotation marks will be printed exactly as it appears in the program followed by the numeric data in line 10 and the result of the computation in line 30.

```
INDIVIDUAL SCORES ARE  82  95  93

THE AVERAGE SCORE IS  90
```

The various uses of the PRINT statement have been presented. Let's take a closer look at the punctuation used in PRINT statements, and how it affects the output. The two primary punctuation marks used in a PRINT statement are the comma and semicolon.

COMMA.—When a comma is used as a separator in a PRINT statement, standard spacing (16 spaces per field) is achieved. When numbers are printed, the first space is reserved for a minus sign. A comma may be used at the end of a PRINT statement to allow the information in the following PRINT statement to be printed on the same line. On some computers, when two commas (,,) are used together they will produce a blank field in printed output.

Here are some examples of what commas do in a PRINT statement.

Example:

Using commas as separators

```
10 PRINT "INDIVIDUAL SCORES"
20 PRINT 82,95,93
30 END
RUN
```

The output from this example would look like this:

```
INDIVIDUAL SCORES
 82            95            93
```

You will notice the output is spaced out into three print zones or fields.

Look at the following example and see what effect a comma at the end of a PRINT statement has on the output.

Example:

Using commas at the end of PRINT statements

```
10 PRINT "THE AVERAGE SCORE IS",
20 PRINT (82+95+93)/3
30 END
RUN
```

The output will look like this:

```
THE AVERAGE SCORE IS    90
```

Here we have two PRINT statements but only one line of output. The comma at the end of line 10 caused the output from line 20 to be printed on the same line as the output from line 10 beginning in the next available print zone.

Let's see what will happen if we use two commas (,,) together in a PRINT statement. (NOTE: This may not work on some computers.)

Example:

Using two commas together

```
10 PRINT "INDIVIDUAL SCORES"
20 PRINT 82,,95,93
30 END
RUN
```

The output will look like this:

```
INDIVIDUAL SCORES
 82                           95            93
```

Here we have a blank field. The second data element started in the third print zone or field instead of the second, because of the two commas used together.

SEMICOLON.—Now let's examine the second punctuation mark used in PRINT statements, the semicolon. When a semicolon is used in a PRINT statement, messages are printed together without any spaces (packed spacing). However, numeric data will be printed with two spaces between each number, one space is reserved for the minus sign. Like the comma, a semicolon used at the end of a PRINT statement causes the information in the next PRINT statement to be printed on the same line. The following examples show the effect a semicolon has on a PRINT statement.

Example:

Using semicolons as separators

```
10 PRINT "INDIVIDUAL SCORES"
20 PRINT 82;95;93
30 END
RUN
INDIVIDUAL SCORES
 82  95  93
```

As seen in this output, the semicolon used as a separator in a PRINT statement causes packed spacing.

Example:

Using semicolons at the end of PRINT statements

```
10 PRINT "THE AVER";
20 PRINT "AGE SCORE IS ";
30 PRINT (82+95+93)/3
40 END
RUN
THE AVERAGE SCORE IS  90
```

As seen in the output from this example, the semicolons caused all the printed output to come out on the same line with the message printed together followed by the numeric data.

Some important points to remember about the PRINT statement are:

- It will print any message enclosed in quotation marks.
- It will print any number or variable.
- It will calculate and print the result of an expression.
- A blank PRINT statement will cause a blank line in your output.
- When a comma is used as a separator, normal spacing (16 spaces per field) is achieved.
- When a comma is used at the end of a PRINT statement, the data in the next PRINT statement is continued on the same line beginning in the next print zone.
- When " " (quote, blank, quote) is used as a print field the computer will skip to the next print field. You are actually telling the computer to print a blank field. On some computers, two commas (,,) together will produce the same results.
- When a semicolon is used as a separator, packed spacing is achieved.
- When a semicolon is used at the end of a PRINT statement, the information contained in the next PRINT statement is continued on the same line.

PRINT statements can become awkward when many computations are to be done or the results of computations are to be used in other calculations. *The results of computations done in a PRINT statement are not stored in the computer's memory;* therefore, a way is needed to store data or the results of computations for later use. The LET statement will do this, since *the results of computations done in a LET statement are stored in the computer's memory.*

LET Statement

One of the more useful BASIC language keywords is **LET**. It is an instruction to the computer to either assign a specified value to a variable name, or to do certain computations and then assign the result to a variable name. The LET statement is not a statement of algebraic equality; rather, it is a definition that assigns a value or a number to a variable.

The LET statement can be used to assign a constant value to a variable name, a variable to a variable name, or the result of an expression to a variable name. Therefore, the LET statement is often referred to as an *assignment statement.*

For example:

Assigning a constant value to a variable name

```
10 LET A = 212
```

In this example the value 212 is assigned to the variable name A and stored in memory with the location name, A. The equal sign should be read as *be replaced by* or more precisely *be assigned the value of*. This does not represent algebraic equality.

Let's examine a program with a LET statement that assigns a constant value to a variable name. Assume we wanted to find the area of three circles, each with a radius of 5, 6, and 7 inches respectively. We know that pi (3.1416) will be used in each computation; therefore, if we assign the constant value of 3.1416 to a variable name (P), we can use the variable name in each computation rather than writing 3.1416 three times. This eliminates repetitive coding. The program could be written this way:

Example:

Assigning a constant value to a variable name

```
10 LET P=3.1416
20 PRINT P*(5**2),P*(6**2),P*(7**2)
30 END
RUN
 78.54          113.0976        153.9384
```

If a value is to be used several times in a program, it can be assigned to a variable name and stored in the computer's memory. When that value is to be used, you can reference it by using the variable name. In the previous example that is what we did, we assigned 3.1416 the variable name of P and each time we wanted to use it, we referenced P rather than write 3.1416 each time.

Since an *expression* is really nothing more than a value represented by algebraic symbols, the LET statement can also be used to assign a variable name the value of an expression as in a formula or equation.

Using the previous example of the circles, let's see how this works.

Example:

Assigning the value of an expression to a variable name

```
10 LET P=3.1416
20 LET A1=P*(5**2)
30 LET A2=P*(6**2)
40 LET A3=P*(7**2)
50 PRINT A1,A2,A3
60 END
RUN
 78.54          113.0976        153.9384
```

In this example, the expressions on the right side of the equal sign were calculated and the results assigned to the variable names A1, A2, and A3. Then in line 50 when we wanted to print the answers we referenced the variable names instead of having to recode the expressions.

Three important points to remember about the LET statement are:

- It assigns a value to a variable name.
- The value may be expressed as a constant, another variable or an expression.
- The value is stored in memory and may be referenced by its variable name.

ARITHMETIC EXPRESSIONS

Arithmetic expressions are composed of a combination of constants, variables, operation symbols, and functions. An expression may be very simple or quite complex, but it will result in a single value. Whether an expression is simple or complex, the calculations must be performed in a specific order. To ensure the computer will correctly evaluate and calculate arithmetic expressions, you have to learn to code them using the rules of BASIC. In order to use arithmetic expressions efficiently, you must be able to evaluate and convert conventional mathematical expressions into proper BASIC expressions.

Arithmetic Operators

Unlike algebra, each arithmetic operator in a BASIC expression must be specified by the inclusion of the appropriate operator symbol. The symbols with their operation are as follows:

Operator Symbol	Operation
**	Exponentiation (an up arrow ↑ is used on some computers)
*	Multiplication
/	Division
+	Addition
–	Subtraction

The symbols associated with each operator are standard in the BASIC language. In mathematics it's all right to write AB to mean multiply A times B. In BASIC you must write A*B since default conditions do not exist. This means if you forget one of the operator symbols, the compiler will not insert it for you; but rather will give you an error message.

Precedence Rule

Many expressions are complex and may have two or more operators; therefore, the computer must have specific rules of precedence to define the order of execution. The computer will perform exponentiation first, then multiplication or division, then addition or subtraction. That is, exponentiation has precedence over addition and subtraction. The following list shows the operator symbols, operation, and their precedence of execution.

Operator Symbol	Operation	Precedence
**	Exponentiation	1
*	Multiplication	2
/	Division	2
+	Addition	3
–	Subtraction	3

If two or more operators of the same precedence appear in an expression, then the order of evaluation is from left to right.

Example:

Given: A = 10, B = 6, and C = 7

```
LET X=A+B-C
```

A and B will be added giving 16, then C will be subtracted giving 9.

```
LET X=A-B+C
```

B will be subtracted from A giving 4, then C will be added giving 11.

Parentheses Rule

There are cases where the precedence rule may cause a problem. For example:

$$y = \frac{a}{b+c}$$

The BASIC expression LET Y = A/B + C would produce undesired results, because A would be divided by B and the result added to C. The solution to this problem is in the use of *parentheses*. If we let parentheses override the order of precedence (but maintain the order of precedence within the parentheses), the result will be satisfactory. Now let's examine the previous example using *parentheses*.

Example:

Using parentheses

$$y = \frac{a}{b+c}$$

```
LET Y=A/(B+C)
```

With the parentheses, B is added to C and the sum of this operation is divided into A, giving the correct result.

Sometimes more than one set of parentheses may be needed to tell the BASIC language in what order to execute the arithmetic operations.

Example:

Parentheses inside parentheses

$$m = \left(\frac{a+b}{c}\right)^2$$

The BASIC expression to accomplish this would be:

```
LET M=((A+B)/C)**2
```

For this expression to give us the correct results, A and B must be added first, then the sum divided by C, and finally that result is squared. When parentheses within parentheses are used, the innermost parentheses will be evaluated first. Addition has a lower precedence than either division or exponentiation; therefore, A + B must be in the inner parentheses. Division has a lower precedence than exponentiation, so (A + B)/C also must be enclosed in parentheses, ((A + B)/C), to ensure it is performed next.

The important thing to remember is the parentheses may be used to override the normal order of precedence. The *parentheses rule* says:

- Computations inside parentheses are performed first.
- If there are parentheses inside parentheses, the operations inside the inner pair are performed first.

CONSTANTS AND VARIABLES

Throughout Chapters 2 and 3 we have been using *constants* and *variables* to refer to numeric values. Like everything else in a programming language, there are rules to be learned about the coding and use of both constants and variables.

In the BASIC language, we have two ways to refer to a numeric value: first, by a numeric-constant representing the value, and second, by an arbitrary name, a numeric-variable name, representing the value.

Numeric-constants

A numeric-constant is a decimal number, whose value does not change. It may be a whole number or have a decimal or fractional part. If it has a fractional part it must be expressed as a decimal number such as 3.5 rather than 3 1/2. If the number gets too large, the system software will convert the number to scientific notation. As described in Chapter 2, this is simply a decimal fraction multiplied by a positive or negative power of ten.

Numeric-variable Name

A numeric-variable name is an arbitrary name that you select, and you and the computer system use to refer to some location in memory. The value contained in that location may change or vary during execution of a program.

Example:

C = 5/9(F – 32) is the formula for converting temperature in Fahrenheit to Celsius (centigrade). In this equation 5, 9, and 32 are all constants; that is, they never change. F is the variable name for temperature in Fahrenheit; its value varies as the temperature goes up and down. C is the variable name for the temperature in degrees Celsius; it refers to the location in memory where the solution to the computation is stored, and its value varies as the value of F varies according to the formula.

A numeric-variable name may be any single letter of the English alphabet (A to Z), or a single letter followed by any single decimal digit (0 through 9). Table 3-1 shows examples of valid and invalid variable names.

Table 3-1.—Numeric-variable Names

Valid	Invalid	Explanation
C	2	must be alphabetic
F2	9K	first character must be alphabetic second character must be numeric
Z5	A83	too many characters, only two characters are allowed
A4	AB	second character must be numeric

String-constants and String-variables

Not all constants and variables are numeric. They may be a series or string of characters such as name ("JOHN DOE") or address ("206 VILLAGE GREEN CIRCLE"). They may be composed of any combination of letters, digits, and special characters and are enclosed in quotation marks. Generally

they are called character strings and may be either a string-constant or a string-variable. In BASIC, variable names for string-variables are any single letter (A-Z) followed by a dollar sign ($).

For example:

String-constant

```
PRINT "CENTIGRADE TEMPERATURE"
```

String-variable

```
LET A$ = "206 VILLAGE GREEN CIRCLE"
```

The important things to remember about *constants* and *variables* are:

- Numeric-constants are whole or decimal numbers that do not change throughout a program.
- Variables, referenced by variable names, may represent numeric values or character strings.
- Variables are used when values may change during execution of a program.
- Variable names for numeric values may be a single alphabetic character (A-Z), or one alphabetic character followed by a single numeric digit (0-9).
- Variable names for character strings are a single alphabetic character (A-Z) followed by a dollar sign ($).
- String constants are character strings that do not change throughout a program.

SUMMARY

Simple problems can be solved with BASIC by using only two or three instructions. These are the END, PRINT and LET statements. To use these effectively, you must know how they work and what rules must be followed in using them.

The **END** statement, which must be the last statement in every BASIC program, has two functions. It indicates to the compiler that there are no more BASIC statements for it to translate and it terminates execution of the program.

The **PRINT** statement is used to instruct the computer to output something either on the terminal or the printer. The standard print line in BASIC is divided into print zones or fields of 16 spaces each.

The two punctuation marks used in PRINT statements are the comma and semicolon. A comma used as a separator in a PRINT statement causes standard spacing and a semicolon causes packed spacing.

Information enclosed in quotation marks in a PRINT statement will be printed exactly as it appears in the program.

The **LET** statement can be used to assign a constant value to a variable name, a variable to a variable name, or the results of an expression to a variable name. The equal sign in a LET statement does not indicate algebraic equality, rather it means be assigned the value of. The value assigned by a LET statement is stored in the computer's memory; therefore, it can be referenced by its variable name.

Both the PRINT and LET statements may contain expressions with arithmetic operations. These arithmetic operations must be specified by the appropriate operation symbol. Should you forget to include the symbol, the computer will not insert it for you, but will give you an error message. Arithmetic operations within an expression are executed in a prescribed order of precedence: exponentiation first; multiplication and division next; and addition and subtraction last. Parentheses are used to alter the normal order of precedence. Operations inside parentheses are performed first. If there are parentheses inside parentheses, the operations inside the inner pair are performed first.

Constants and variables are used to refer to numeric values or character strings. A constant is a whole or decimal number or character string whose value does not change. A variable name is an arbitrary name you select and you and the computer use to refer to a value stored in the computer's memory. This value may vary during execution of the program, but can contain only one value at a time.

CHAPTER 3

EXERCISES

1. You have been jogging regularly for six months and you want to know how many miles you have jogged in the six month period. Write a program that will total your miles jogged and print a heading that says:

   MILES JOGGED IN SIX MONTHS ARE

   Your miles for each month are: 182,178,174,170,187,183

2. Modify the program from question one to include the computation of average miles jogged per month and a heading that says:

   AVERAGE MILES JOGGED PER MONTH IS

   Leave a blank line between headings.

3. Write a program to convert Fahrenheit temperatures to Celsius and print the results. The Fahrenheit temperature readings are: 76, 88, and 96. The formula C = (F − 32)*5/9.

4. Which of the following are valid numeric-variable names? For those that are not valid, what is wrong with them?

A. R	E. A2
B. R1	F. AA
C. 9	G. M52
D. 9L	H. A$

5. For the following expressions write the BASIC statements to solve the expression.

   A. $a = \frac{x+y}{z}$

   B. $b = \frac{x^2}{y^2}$

   C. $c = \frac{(x+y)^2}{z}$

   D. $d = \left(\frac{x^2}{y+z}\right)^2$

CHAPTER 3

EXERCISE ANSWERS

1.
```
10   LET M = 182 + 178 + 174 + 170 + 187 + 183
20   PRINT "MILES JOGGED IN SIX MONTHS ARE ";M
30   END
RUN
MILES JOGGED IN SIX MONTHS ARE  1074
```

2.
```
10   LET M = 182 + 178 + 174 + 170 + 187 + 183
20   LET A = M/6
30   PRINT "MILES JOGGED IN SIX MONTHS ARE ";M
40   PRINT
50   PRINT "AVERAGE MILES JOGGED PER MONTH IS ";A
60   END
RUN
MILES JOGGED IN SIX MONTHS ARE  1074

AVERAGE MILES JOGGED PER MONTH IS  179
```

3.
```
10   LET F = 76
20   LET F1 = 88
30   LET F2 = 96
40   PRINT (F - 32)*5/9,(F1 - 32)*5/9,(F2 - 32)*5/9
50   END
RUN
 24.44          31.11          35.55
```

4. valid numeric-variable names

   A,B,E

   invalid numeric-variable names

   C. must be alphabetic

   D. first character must be alphabetic

   F. second character must be numeric

   G. variable name can only be two characters

   H. valid only for string-variables

5. A. LET A=(X+Y)/Z

   B. LET B=X**2/Y**2

   C. LET C=(X+Y)**2/Z

   D. LET D=(X**2/(Y+Z))**2

CHAPTER 4

MORE ON SOLVING PROBLEMS WITH BASIC

READ, DATA, RESTORE, INPUT, GOTO, IF-THEN, ON-GOTO, and Loops

In Chapter 3 we discussed the LET statement, which is one way of introducing data into a program. However, this can be very awkward if you want to run a program more than once with new input data. It would be better to develop generalized programs that can be run again and again with different data. This can be done with the READ and DATA statements or with the INPUT statement. With the READ and DATA statements, the data is stored in the program. With the INPUT statement, the data is entered during program execution.

READ and DATA Statements

Although the data is stored with the program, the **READ** and **DATA** statements allow you the flexibility to handle data independently from the steps used to solve the problem. It is much easier to change a few DATA statements each time you want to run new data, than it is to locate and change all the LET statements. This will be even easier if you adopt the recommended programming practice of placing all the DATA statements together, either at the beginning or end of the program.

The DATA statement is used in conjunction with the READ statement. The data specified by DATA statements is stored in a data list in the computer's memory. READ statements cause data to be read from the data list in the DATA statement and assigned the variable names specified in the READ statements.

Example:

```
10  DATA 100,200,1000,5000
20  READ A,B,C,D
```

When a READ statement is executed, the values in the data list are assigned consecutively to the variable names in the READ statement. The first variable name (A) in the READ statement is assigned to the first value (100) in the DATA statement, the second variable name (B) to the second value (200), and so on. At any given time, there is one value at the top of the data list and after it is used, the next one comes to the top of the list.

Each READ statement causes as many values to be taken from the data list as there are variable names in the READ statement. When you are reading

several variables, care should be taken to ensure that the number of variables assigned in one or more READ statements is equal to the number of values assigned in one or more DATA statements. An excess of data in DATA statements is ignored by the computer, but insufficient data will result in an error message, and the computer will halt execution of the program.

Example:

Data list too short

```
10 READ A,B,C
20 DATA 9,8
```

When the READ statement is executed, you will get an insufficient (out of) data error message because there is no data element for variable C.

Example:

Data list too long

```
10 READ A,B,C
20 DATA 9,8,7,6,5
```

When the READ statement is executed, you will NOT get an error message, rather the values 6 and 5 in the DATA statement will be ignored.

All variables in READ statements, and all values in DATA statements are separated by commas. Although DATA statements can be placed anywhere in the program, it is a good programming practice to place them all together either at the beginning or end of the program. This will make them easier to find and to change when you want to run the same program using different data.

If you have programmed in another language, you'll notice that BASIC is different because the size of your data elements do not have to be defined. In COBOL your data elements must be defined using a PICTURE clause, and in FORTRAN they must be defined using a FORMAT statement. BASIC accepts the data elements just as they appear in the DATA statement.

When you are coding DATA statements, you should ensure that the values in the DATA statements are in the same order as the variable names specified by the READ statements. DATA statements are used in the same order as they appear in the program.

Now let's see how the READ and DATA statements work in solving a problem.

Suppose you have been in the "Run for your Life" program for 12 months. The first 8 months you ran 875 miles. During the next 4 months your

monthly mileages were 122, 128, 125, and 118. You want to write a program that will compute and print the number of miles you ran in the 12 month period and your monthly average. A program to do this could be written this way:

```
01  REMARK THIS PROGRAM COMPUTES TOTAL
02  REMARK MILES RUN AND AVERAGE MILES
03  REMARK RUN PER MONTH
10  DATA 875
20  DATA 122,128,125,118
30  READ A
40  READ B,C,D,E
50  LET T = A+B+C+D+E
60  LET A1 = T/12
70  PRINT "MILES RUN IN 12 MONTHS ARE ";T
80  PRINT "AVERAGE MILES RUN EACH MONTH IS ";A1
90  END

RUN


MILES RUN IN 12 MONTHS ARE     1368
AVERAGE MILES RUN EACH MONTH IS     114
```

Note the use of multiple READ and DATA statements, lines 10 through 40. This has no significance except to show that multiple READ and DATA statements can be used.

When this program is executed, the values in the DATA statements, lines 10 and 20, are assigned consecutively to the corresponding variable names in the READ statements lines 30 and 40. The number 875 is assigned to A, 122 to B, 128 to C and so on.

Next the program totals the miles run in 12 months and calculates the average miles per month. It then prints a heading for total miles run followed by the total, and a heading for average miles run followed by the average. Each time you want to run the program using new data, all you have to do is change the DATA statements, in lines 10 and 20.

RESTORE Statement

In most programs, values in data lists are read and processed only once. However, there are times when it is desirable to re-read data within a program; you can use the **RESTORE** statement to reset the pointer to the top of the data list. In this way, the next READ statement executed starts reading data from the top of the data list. The first variable in the READ statement is assigned the first value in the first DATA statement.

Example:

Program		Data List And Variable Assignment	Pointer Position After Statement Execution
10	**DATA 5,10,15,20,25,30,35**	5,10,15,20,25,30,35	1st Data Element
20	**READ A,B,C**	5,10,15,20,25,30,35 A B C	4th Data Element
30	**RESTORE**	5,10,15,20,25,30,35	1st Data Element
40	**READ W,X,Y,Z**	5,10,15,20,25,30,35 W X Y Z	5th Data Element
50	**PRINT A;B;C;W;X;Y;Z**		
60	**END**		

RUN

5 10 15 5 10 15 20

The first READ statement, line 20, assigns 5,10, and 15 to A,B,C respectively. The RESTORE statement, line 30, then resets the pointer to the top of the data list. The second READ statement, line 40, assigns the first data element, 5, to variable W; the second, 10, to X; the third, 15, to Y; and the fourth, 20, to Z. The last three values in the data list, 25, 30, and 35 will not be assigned.

INPUT Statement

Another way to provide data to a program is using the **INPUT** statement. Input statements are used if data is to be supplied during program execution. They can be placed anywhere in a program prior to the statements that need the data. When the computer encounters an INPUT statement during program execution, it displays a question mark, stops, and waits for the necessary data to be entered. Once the data has been entered, execution proceeds.

For example, if you want to supply values for the variables X and Y in a program, you include the statement:

```
10 INPUT X,Y
```

This statement must be executed before the first statement that is to use either of the two variables. When this INPUT statement is executed, the computer displays a question mark on the terminal and waits for the values of X and Y to be input. You then enter the appropriate two numbers separated by a comma.

Since the INPUT statement signals the need for data with only a question mark, it is good programming practice to precede each INPUT statement with a PRINT statement to remind you what values are to be input and in what order. This is particularly important in a program with several INPUT statements.

The INPUT statement specifies one or more variable names which must be separated by commas. There must not be a comma at the end of the statement. Only variable names are used in the INPUT statement, no values may be placed in the INPUT statement. The INPUT statement is like a READ statement because it is a way to assign values to variable names. However, the data is entered during program execution, rather than supplied in a DATA statement in the program. You must specify the number of values and in what order they are to be entered by listing one variable name for each value to be entered. If you enter too few values the computer will respond with question marks until the required number of values have been entered. If you enter too many values, the excess values will be ignored.

Data provided using the INPUT statement is entered during program execution but *is not* saved as part of the program once the program has been run.

INPUT statements are easy to use, interactive, and conversational. In essence, you are carrying on a conversation with the computer during program execution. You don't have to change any part of the program to use new data. You might ask, "Why would anyone ever want to use READ and DATA statements?" Suppose you had a larger program with several INPUT statements containing many variables. One mistake in entering the data would mean the program must be run again, reentering the data and ensuring that it is all entered correctly. When using READ and DATA statements, a mistake may be corrected more easily, by changing only the DATA statement that is in error and running the program again.

The following program shows how to use the INPUT statement to enter the values needed to compute your monthly payment on a loan: given the annual interest rate, number of years of the loan, and amount of the loan.

Example:

```
01    REMARK THIS PROGRAM COMPUTES MONTHLY
02    REMARK LOAN PAYMENT
10    PRINT "INTEREST,YEARS,LOAN"
20    INPUT I,Y,L
30    LET N3 = 1+I/12
40    LET N4 = N3**(12*Y)
50    LET N2 = (I/12)*N4
60    LET N1 = L*N2
70    LET D = N4-1
80    LET P = N1/D
90    PRINT P
100   END
```

After you have entered the system command RUN, the message

```
INTEREST,YEARS,LOAN

?
```

will appear on your terminal. You now enter the values for each variable:

```
.12,20,15000
```

.12 is for 12% interest rate, 20 for number of years, and 15000 for the amount of the loan, $15,000. Press the return key to enter the values into the computer. The program will continue execution with statement number 30, do the calculations, and print the result, $165.16.

CONTROL STATEMENTS

Up to this point, the BASIC statements we have used have been executed sequentially. That is, after one statement has been executed, the statement immediately following it (the one with the next higher line number) will be executed until the program ends. Now we will discuss the statements which may be used to alter this sequence of execution and some of the reasons why this may be desirable.

This change in the normal sequence of execution is called *transfer of control* or *branching*. There are two categories of transfer of control statements. They are *conditional* and *unconditional. Conditional transfer of control* is achieved through the evaluation of a condition. Relational symbols are used in evaluating the condition.

For example:

```
20 IF X>2 THEN 50
```

If the condition (X>2) proves true, the program branches to line 50; if the condition proves false, processing continues sequentially through the program. *Unconditional transfer of control* is not dependent on any condition. It is achieved by the control statement, GOTO, which specifies the line number to which control is to be transferred.

For example:

```
40 GOTO 99
```

When this statement is executed control will be unconditionally transferred to line 99.

GOTO Statement (Unconditional)

Using the unconditional GOTO statement is one method of interrupting the sequential execution of program statements. You can use it to branch (transfer control) to any part of a program during execution.

In the first part of the chapter, we discussed how to use the READ, DATA and INPUT statements to introduce data into a program. If we wanted to run several sets of data through a program, it would be necessary to rerun the program for each new set of data. By putting a GOTO statement in a program, we can transfer control back to the beginning of the program to process additional data. This creates a loop. A *loop* is any sequence of statements that is to be repeated some specified number of times, or until a particular condition is met.

Example:

Loop using GOTO, READ and DATA

```
01    REMARK THIS PROGRAM CONVERTS
02    REMARK FAHRENHEIT TO CELSIUS
10    DATA 96,93,88,102,85
20    DATA 82,94,98,212,32
30    PRINT "FAHRENHEIT", "CELSIUS"
40    READ F
50    LET C = (F-32)*(5/9)
60    PRINT F,C
70    GOTO 40
80    END
RUN
FAHRENHEIT      CELSIUS
 96              35.55555555556
 93              33.88888888889
 88              31.11111111111
 102             38.88888888889
 85              29.44444444445
 82              27.77777777778
 94              34.44444444445
 98              36.66666666667
 212             100
 32              0
```

The GOTO statement in line 70 creates the loop. That is, it tells the program to branch back to the READ statement in line 40, which assigns the next data element to variable F, performs the calculations, prints the results, and repeats this process until all the data elements have been used. The first time through the loop, F will equal 96, the second time F equals 93 and so on. When all the data elements have been read, you get an "out of data" error message. Later, we will discuss how to prevent this, using other control statements.

Study the loop and become familiar with how a loop works. Note that the PRINT statement in line 30 which prints the heading is outside of the loop. One major point to remember in creating a loop is that you only want to include functions that are to be performed repetitively inside the loop. Do not include statements that are to be performed only once during program execution. If we had included line 30 inside the loop, a heading would have been printed each time the loop was executed.

Now let's examine how a loop with an INPUT statement can be used to introduce data into a program.

Example:

Loop using GOTO and INPUT

```
01  REMARK THIS PROGRAM CONVERTS
02  REMARK FAHRENHEIT TO CELSIUS
10  PRINT "ENTER FAHRENHEIT TEMPERATURE"
20  PRINT "FAHRENHEIT" , "CELSIUS"
30  INPUT F
40  LET C = (F-32)*(5/9)
50  PRINT F,C
60  GOTO 30
70  END
```

Using the GOTO statement in this program will cause the computer to query the user for more input after each entry has been processed. Note that lines 10 and 20 are outside of the loop. Line 10 is a prompt used to specify what to input, and line 20 is the heading which we wouldn't want printed each time. This program will stay in this loop and continue to query the user because we did not provide an exit from the loop. The statements we can use to control (exit) a loop will be discussed next.

IF-THEN Statement (Conditional)

Perhaps one of the most powerful statements in BASIC's set of instructions is the **IF-THEN** statement. It is a statement of conditional branching; that is, it conditionally changes the consecutive order of execution of statements depending on the outcome of some test or relationship. Relational symbols and expressions are used in constructing IF-THEN statements. If the relationship proves true, control is transferred to a specified line number. If the relationship proves false, the program proceeds to the next sequential statement. The following example shows the fundamental concept of how an IF-THEN statement works and another use of the GOTO statement.

Example:

```
10  INPUT A,B
20  IF A = B THEN 50
30  PRINT "A DOES NOT EQUAL B"
40  GOTO 60
50  PRINT "A EQUALS B"
60  END
```

In this example, the test of the relationship between A and B is made at line 20 with a conditional branch to line 50 if A is equal to B. If A is not equal to B, the next statement executed is at line 30. In the latter case we do not want the program to execute the statement at line 50. To avoid this, we write a GOTO statement at line 40, which unconditionally transfers control to line 60.

As you can see, the IF-THEN statement causes a change in execution sequence only on meeting a particular condition of data. In contrast, the GOTO statement, line 40 causes a change in sequence every time it is executed.

Loop Control with IF-THEN

We have seen the IF-THEN statement used as a tool for branching to a different area of a program; now let's examine its use in controlling a loop. Remember, a loop is a sequence of statements that is to be repeated some specified number of times, or until a particular condition is met. The number of times through the sequence can be a constant number fixed at the time a program is written, or it can be a variable number depending upon some condition existing at the time of program execution. You, as the programmer, are responsible for initiating the loop and controlling how many times the program will execute the loop.

Consider the task of calculating all powers of 2 from 2**1 through 2**10. (2**10 is the notation BASIC uses for "2 raised to the tenth power,"

exponentiation.) The following procedure shows how to do this, using the loop technique:

(1) Set K = 1

(2) Compute 2**K

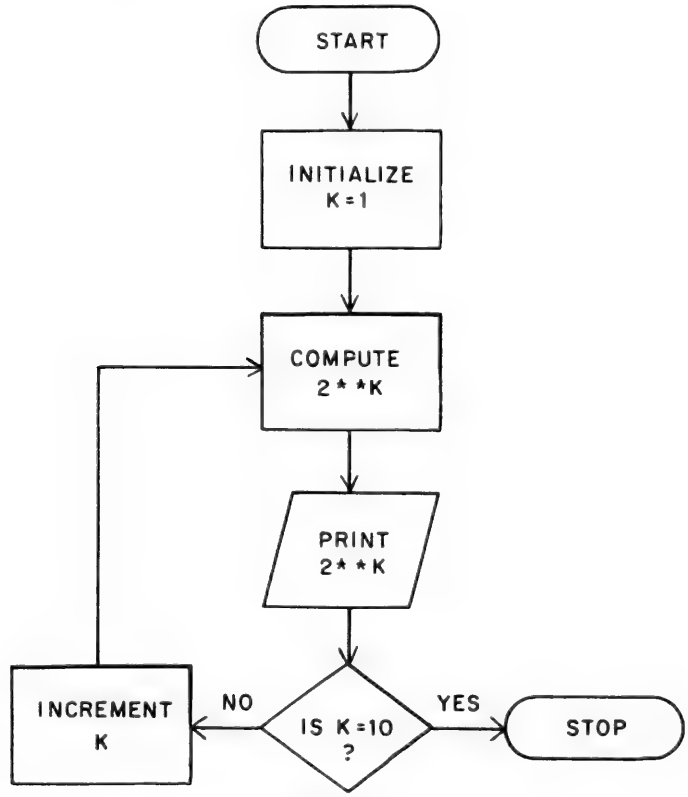
(3) Print 2**K

(4) If K is equal to 10, Stop

(5) Add 1 to K

(6) Go back to step (2)

The flowchart for this procedure would look like this:



Flowchart

We've used the variable K as the exponent so we can vary its value from 1 to 10 during program execution. K will initially be set to 1. Then 2**K will be computed and printed. Since K is not equal to 10 the program will take the "NO" path, add 1 to K and go back to repeat step 2. Now 2**K (which is 2**2) is computed and printed, then the values for 2**3, 2**4 ..., 2**10

are all computed and printed. However, after 2**10 has been printed, K will equal 10 and the program will take the "YES" path, causing the program to halt.

The following is a program for the flowchart procedure.

```
01  REMARK THIS PROGRAM COMPUTES
02  REMARK THE POWERS OF 2 FROM 1 to 10
10  LET K = 1
20  LET X = 2**K
30  PRINT X
40  IF K = 10 THEN 70
50  LET K = K+1
60  GOTO 20
70  END
```

Note that line number 40 transfers control to line 70 only when K = 10. Line 60 changes the sequence every time it is executed. Line 50 shows a method to accomplish counting. This statement means that one is added to the current value of K, and the result is assigned to K as its new value. When using a variable as a counter, you should initialize the counter either to zero or some other specified value. In this example the value of K was initially set to 1. This eliminates the possibility of accumulating erroneous totals.

When the program is run, the output will look like this:

```
 2
 4
 8
 16
 32
 64
 128
 256
 512
 1024
```

There are times when the amount of data we want to process will vary. For example, you might want to keep track of the number of miles per gallon you are getting on your car. A program to solve this problem could be written this way:

```
01   REMARK THIS PROGRAM COMPUTES
02   REMARK AVERAGE MILES PER GALLON
10   DATA 485,16,520,17.5,450,15,360,13,0,0
20   LET M1 = 0
30   LET G1 = 0
40   READ M,G
50   IF M = 0 THEN 90
60   LET M1 = M1+M
70   LET G1 = G1+G
80   GOTO 40
90   LET A = M1/G1
100  PRINT "AVERAGE MILES PER GALLON IS ";A
999  END
```

The counters M1 and G1 are initialized to zero in lines 20 and 30. They will be used to accumulate total miles (1815) and total gallons (61.5) which will be used to compute the average miles per gallon (29.51).

In this program, we are using a loop (lines 40-80) to read and process sets of data. Look at the READ statement in line 40. On the first pass through the loop, the variable name M will be assigned 485, and G assigned 16. The second pass M will be assigned 520, and G assigned 17.5, and so on until M = 0. Zeros are placed after the data to indicate end of data. They serve as dummy (artificial) data elements which can be tested for in the program to determine when all data has been processed. This allows any number of sets of data to be included in DATA statements. Because the data is read in pairs, two zeros were used to avoid getting an "out of data" error message.

The IF-THEN statement in line 50 tests for M = 0. This controls the loop and transfers control (exits the loop) to line 90 when all the input data has been read and processed.

In this way the number of times the loop is executed does not have to be predetermined as in the previous "powers of two" problem where we set the number at 10. This also prevents an "out of data" error message which would be catastrophic in this program. The formula to compute average miles per gallon and the PRINT statement to print the results are outside the loop;

therefore, an "out of data" error condition would prevent the program from ever reaching lines 90 and 100.

This program differs from the "Loop using GOTO, READ and DATA" example in the previous section. The statements which did the calculations and printing in that example were inside the loop. Therefore an "out of data" condition was not a catastrophe because all the data had been processed and printed before an "out of data" condition was reached.

ON-GOTO Statement (Conditional)

The IF-THEN statement allows the computer to follow one of two possible paths. The **ON-GOTO** differs from the IF-THEN statement—it allows many branches or alternative paths. Several IF-THEN statements could be combined to produce the same results; however, the ON-GOTO statement can be more efficient. It conditionally transfers control to one of several lines, depending on the value of an expression in the ON-GOTO statement when it is executed.

In an ON-GOTO statement, the expression is evaluated and the result truncated to obtain an integer, whose value is then used to select a line number from the list following the GOTO. If the value of the integer is 1, the computer goes to the first line number; if it is 2, the computer goes to the second line number and so forth.

Example:

```
50 ON X GOTO 90,100,110
```

This statement will transfer control to:

- line 90 if X is greater than or equal to 1 but less than 2.
- line 100 if X is greater than or equal to 2 but less than 3.
- line 110 if X is greater than or equal to 3 but less than 4.

If values of X other than 1, 2, or 3 are encountered, you would get an error message on some computers; on others, the statement would be ignored.

Suppose you wanted to write a program to determine what your return would be on savings deposited in an account paying the following annual interest rates.

4.0% paid on savings of $1000-1999

4.5% paid on savings of $2000-2999

5.0% paid on savings of $3000-4999

The following program asks you to enter the amount saved, between $1000 and $4999, then uses the ON-GOTO to determine what interest rate to use in calculating the amount of interest earned.

```
10    REMARK THIS PROGRAM COMPUTES INTEREST
20    REMARK EARNED ON SAVINGS
30    PRINT "ENTER AMOUNT SAVED, 0 IF NO MORE DATA"
40    INPUT X
50    IF X = 0 THEN 160
60    ON X/1000 GOTO 90,110,130,130
70    PRINT "VALUE OUT OF RANGE"
80    GOTO 40
90    LET I = .04*X
100   GOTO 140
110   LET I = .045*X
120   GOTO 140
130   LET I = .05*X
140   PRINT "ANNUAL INTEREST ON $";X;" IS $";I
150   GOTO 40
160   END
```

If 1500 were entered, line 60 would transfer control to line 90 where the interest rate of 4% is used to calculate the interest earned in a year. If 3500 were entered, line 60 would transfer control to line 130. Likewise, if 4800 were entered, it would also transfer control to line 130.

You probably noticed a zero (0) was to be entered when there was no more data to be processed, and that line 50 tested for 0. If X equals 0, control transfers to line 160 and stops. This is another example of testing a variable condition to determine when to exit a loop.

Note, also, the statements in lines 70 and 80, these act as an error routine. If you enter a value other than 1000 through 4999, line 70 prints the message, "VALUE OUT OF RANGE" and line 80 transfers control back to line 40 for you to enter another value.

Sample Problem Using Control Statements

Now let's examine a program which contains an example of each type of control statement discussed: **GOTO**, **IF-THEN**, and **ON-GOTO**.

In this example we will be converting temperatures from Fahrenheit to Celsius and vice versa. Suppose we want to write a program to solve this problem, and to warn us if any temperature we convert to Celsius is over 100° Celsius. The first step in solving the problem is to formally define the

problem to be sure we know exactly what problem we want to solve, and what its inputs and outputs are to be. The PROBLEM DEFINITION could be written as shown in figure 4-1.

The procedure outlined in the programming flowchart and coded program (figure 4-2) will solve our problem.

Study the flowchart and the program. To help, the statement numbers have been placed on the flowchart above each corresponding flowchart symbol.

Line 10 will print a message prompting you to input the temperature you want converted. The program will then ask which conversion you want; you respond with 1 for F to C, or 2 for C to F. Line 60 will evaluate your response to determine whether to branch to line 90 or line 150 (conditional branch). If you entered anything other than a 1 or 2 at line 50, you'll get a message and the program will branch back to line 50 because of the GOTO 50 in line 80. If you entered a 1, control will be transferred from line 60 to line 90, where the value you input at line 20 will be converted to a Celsius value. Line 100 will take this value and check it to determine if it is greater than 100. If it is, control is transferred to line 130 where the value for C and a message is printed. Line 140 is executed next. It is a GOTO statement (unconditional branch) which transfers control to line 170. At this point you are given the option to terminate the program.

Let's assume you enter a Y (line 180), the program will branch back to line 10. Now you may enter another value to be converted. This time enter a 2 at line 50. The ON-GOTO statement, line 60, will transfer control to line 150, which will convert the value entered at line 20 into a Fahrenheit value, and print the value of F and a message. Once this is done, line 170 will again be executed.

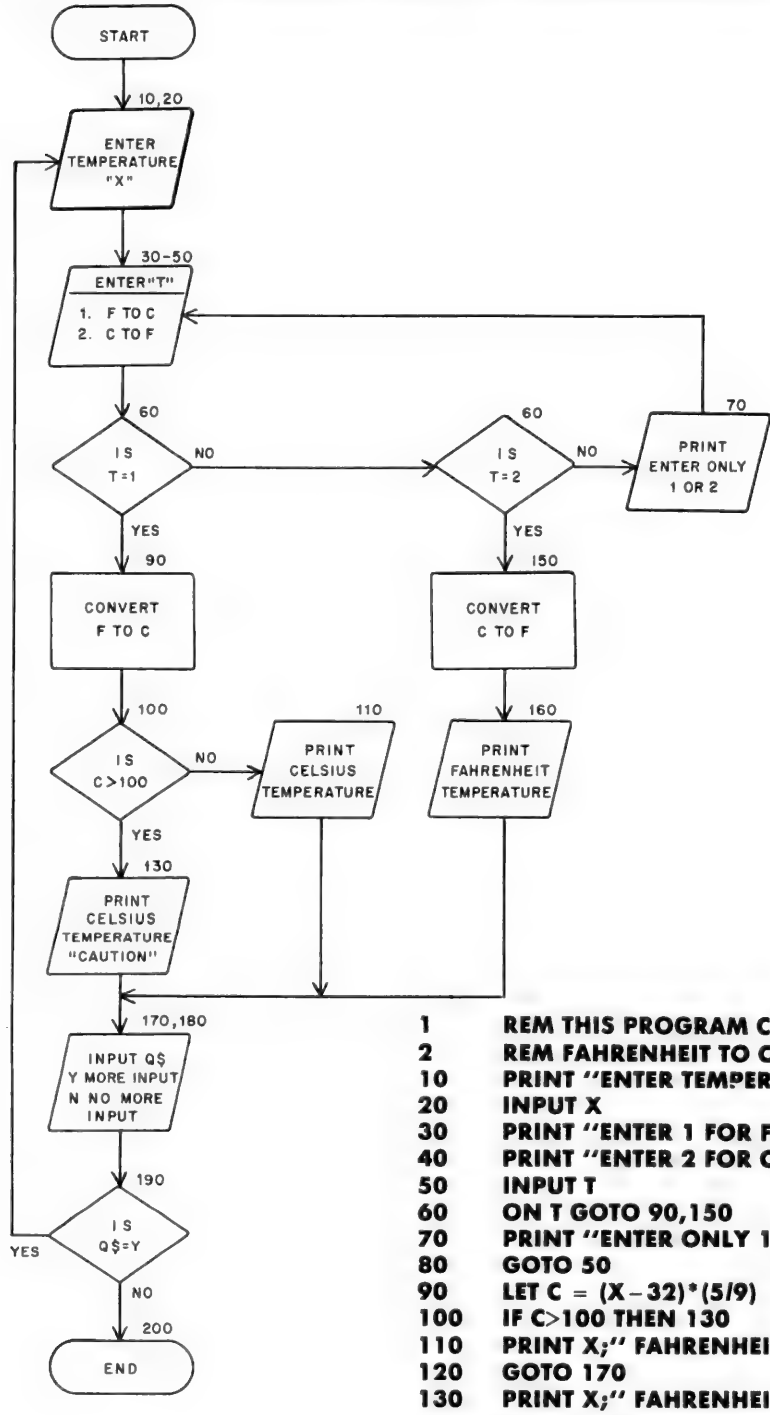
PROBLEM DEFINITION

TEMPERATURE CONVERSION—This program is to convert temperatures from Fahrenheit (F) to Celsius (C) and from Celsius to Fahrenheit. The formulas to do this are: C=(F−32)*5/9 and F=9/5*C+32. The program is also to print a warning beside any Celsius value greater than 100 degrees Celsius. The program is to contain a loop to allow for repeated executions to process more input.

INPUT: The input for this program is to be either a Fahrenheit or Celsius value which is to be input as variable X. Another variable, T, will be used to indicate the type conversion you wish to do; Fahrenheit to Celsius or Celsius to Fahrenheit. String variable Q$ is to be used to specify whether there is more input or not. All values for these variables will be entered into the system via the terminal at the time of program execution.

OUTPUT: For each temperature entered, print it and its converted value with a caption to indicate whether it is Fahrenheit or Celsius and for any Celsius value greater than 100 degrees, print the word "CAUTION" beside it.

Figure 4-1.—Temperature Conversion Problem Definition.




```
1       REM THIS PROGRAM CONVERTS TEMPERATURES,
2       REM FAHRENHEIT TO CELSIUS AND VICE VERSA
10      PRINT "ENTER TEMPERATURE"
20      INPUT X
30      PRINT "ENTER 1 FOR FAHRENHEIT TO CELSIUS"
40      PRINT "ENTER 2 FOR CELSIUS TO FAHRENHEIT"
50      INPUT T
60      ON T GOTO 90,150
70      PRINT "ENTER ONLY 1 OR 2"
80      GOTO 50
90      LET C = (X-32)*(5/9)
100     IF C>100 THEN 130
110     PRINT X;" FAHRENHEIT = ";C;" CELSIUS"
120     GOTO 170
130     PRINT X;" FAHRENHEIT = ";C;" CELSIUS, *CAUTION*"
140     GOTO 170
150     LET F = 9/5*X+32
160     PRINT X;" CELSIUS = ";F;" FAHRENHEIT"
170     PRINT "ENTER Y IF MORE INPUT, N IF NO MORE INPUT"
180     INPUT Q$
190     IF Q$ = "Y" THEN 10
200     END
```

Figure 4-2.—Temperature Conversion Flowchart and Program.

Now take some values you know the Fahrenheit and Celsius equivalent of, and trace them through the program. See what paths would be taken and determine if the program is correct. This procedure, discussed in Chapter 1, is known as "desk checking." You might start with 32 degrees Fahrenheit. Trace it through the program to be sure you get 0 degrees Celsius. Next take 212 degrees Fahrenheit; did you get 100 degrees Celsius? Try 100 degrees Celsius; did you get 212 degrees Fahrenheit? If you take 213 degrees Fahrenheit, you should get 100.555 degrees Celsius and the message, CAUTION; did you?

Once you have tested with known values to test the various program paths, you can be reasonably sure that the program will run and produce accurate results.

SUMMARY

The **READ** and **DATA** statements can be used to introduce data into a program. READ statements cause data to be read from a data list specified in a DATA statement and assigned the variable names specified in the READ statement. When coding DATA statements, be sure the values in the DATA statements are in the same order as the variable names specified by the READ statements.

Data may also be introduced into a program at the time the program is executed by using the **INPUT** statement. The INPUT statement specifies one or more variable names; each must be separated by a comma. It is good programming practice to precede INPUT statements with a PRINT statement to print a message to remind you what data is to be input and in what order. Data introduced into the program using the INPUT statement is not retained by the program after the program has been executed.

Transfer of control statements may be used in a program when you do not want to execute statements in sequence, or when you want to execute a group of statements repeatedly, forming a *loop*. The unconditional transfer of control statement, **GOTO**, is used when you always want to alter the normal sequence, either to bypass some statements or to branch back to the beginning of a loop. The conditional transfer of control statements, **IF-THEN** and **ON-GOTO** are used when you only want to change the normal sequence of execution if given conditions exist. These can be used to control a loop or to branch to selected statements.

CHAPTER 4

EXERCISES

1. Study the following statement:

   75 ON X GOTO 50,90,130,50

   A. What statement number will control be transferred to, if the integer value of X is 2?
   B. What integer value of X will cause control to be transferred to statement number 130?
   C. What, if any, integer value(s) of X will cause control to be transferred to line number 50?

2. Write an ON-GOTO statement to branch to line 60 if age (A) is 10-19, to line 80 if age is 20-39, to line 150 if age is 40-59 and to line 180 if age is 60-65. (Assume there are no ages less than 10 or greater than 65.)

3. Refer to the program in figure 4-2.

   A. What condition would cause lines 70 and 80 to be executed?
   B. If the value 216 is entered as variable X and you ask for F to C conversion, what are the line numbers of the statements that would be executed?
   C. What is the purpose of statements 170-190?

4. Given a monthly beginning bank balance and a list of transactions (deposits and checks), write a program to list deposits and checks, and compute and print the balance after each transaction. If ending balance is $500.00 or less, print a message that says A SERVICE CHARGE OF $4.00 HAS BEEN DEDUCTED FROM YOUR ACCOUNT. YOUR FINAL BALANCE IS _____.

   The beginning balance is to be entered at the time of program execution, transactions are to be stored in the program. Use zero (0) to indicate end of data. Use a loop to read and process the transactions.

Beginning balance	$2450.25
Transactions	50.00 check
	37.40 check
	320.45 deposit
	100.20 check
	25.00 deposit
	85.35 check
	250.00 check
	200.00 deposit

CHAPTER 4

EXERCISE ANSWERS

1. A. Control will be transferred to line 90

   B. An integer value of 3 will transfer control to line 130

   C. An integer value of 1 or 4 will transfer control to line number 50. Statement numbers may be repeated in line number specifications in ON-GOTO statements.

2. ON A/10 GOTO 60,80,80,150,150,180

   In this example the value of the variable is divided by a specified number to get the integer value.

3. A. An entry other than a 1 or 2 would cause lines 70 and 80 to be executed.

   B. The line numbers of the statements that would be executed following line number 50 are 60,90,100,130,140,170, and the program will stop at 180 and wait for a response.

   C. The purpose of statements 170-190 is to control the loop. Statement 190 tests the value of the string variable, Q$; if it is equal to Y (more input), then control is transferred back to statement number 10 to request and process the next input. If Q$ is equal to N (no more input) execution of the program is terminated by statement number 200.

4. This is one way the program could be written to solve the problem.

```
10    DATA -50.00,-37.40,+320.45,-100.20,+25.00
11    DATA -85.35,-250.00,+200.00,0
20    PRINT "ENTER BEGINNING BALANCE"
30    INPUT B
40    PRINT "YOUR BEGINNING BALANCE IS $";B
50    PRINT "CHECKS/DEPOSITS","BALANCE"
```

```
60      READ C
70      IF C = 0 THEN 110
80      LET B = B+C
90      PRINT C,B
100     GOTO 60
110     IF B>500 THEN 150
120     PRINT "A SERVICE CHARGE OF $4.00 HAS BEEN ";
130     PRINT "DEDUCTED FROM YOUR ACCOUNT."
140     LET B = B-4.00
150     PRINT "YOUR FINAL BALANCE IS     $";B
999     END
RUN
ENTER BEGINNING BALANCE
?
YOUR BEGINNING BALANCE IS $ 2450.25
CHECKS/DEPOSITS         BALANCE
-50                     2400.25
-37.4                   2362.85
 320.45                 2683.3
-100.2                  2583.1
 25                     2608.1
-85.35                  2522.75
-250                    2272.75
 200                    2472.75
YOUR FINAL BALANCE IS     $ 2472.75
```

CHAPTER 5

WRITING MORE EFFECTIVE AND EFFICIENT PROGRAMS

FOR-NEXT, STEP, DIM, GOSUB, RETURN, Arrays, and Nested Loops

By now you realize the power of a computer is in its capability to do many computations over and over on different data. While a great deal of detail and precision is required when writing a program, once written it can be used again and again. You can see that as problems increase in size and complexity, programming becomes more tedious and time consuming, especially if you are limited only to the keywords presented in Chapters 3 and 4. Fortunately, there are additional keywords:

- FOR-NEXT—for simplifying loops
- DIMENSION and subscripted variables—for processing data in tables (arrays of one or more dimensions)
- Predefined functions—for computing commonly used mathematical functions
- DEF—for defining your own functions
- GOSUB and RETURN—to allow the use of subroutines
- STOP—to terminate program execution anywhere in a program

SIMPLIFYING LOOPS USING FOR-NEXT

In Chapter 4, we saw that loops can be very useful when you have a series of statements you wish to repeat a number of times. BASIC provides two additional keywords that make some loops even easier to construct. They are **FOR-TO** and **NEXT**.

A FOR-NEXT loop always begins with a FOR-TO statement and always ends with a NEXT statement. The complete loop is comprised of all statements included between the FOR-TO and the NEXT statements.

Example:

```
45  FOR M = 1 TO 5
    .
    .
    .
75  NEXT M
```

This loop will consist of all statements, from statement number 45 through statement number 75, and it will be executed 5 times.

The FOR-TO statement specifies how many times the loop is to be executed. It must be the first statement in the loop. The FOR-TO statement has a numeric variable, called the running variable, whose value changes each time the loop is executed. The number of executions is determined by specifying the initial and final values for the running variable. In the example, M is the running variable; 1 is the initial value of M, and 5 is the final value of M. Each time through the loop, M is increased by 1. When M equals 5 the program exits the loop.

The NEXT statement consists of a statement number, followed by the keyword, NEXT, followed by a running variable name. This running variable must be the same as the running variable that appears in the corresponding FOR-TO statement.

FOR-TO Statement

A typical FOR-TO statement would look like this:

Example:

```
65  FOR M = 1 TO 36
```

In this example, M is the running variable. The first time the loop is executed, M will be assigned a value of 1. M will increase by 1 each time the loop is executed, until M has reached its final value of 36. The loop will be terminated once M has exceeded its final value of 36. The loop in the example will be executed 36 times.

The running variable will always increase by 1, if the FOR-TO statement contains no instructions telling it to do otherwise. However, we can increment the running variable by some value other than 1 if we wish. This can be done by the addition of a **STEP** clause to the FOR-TO statement.

Suppose we want to execute a loop 50 times, and we want the running variable to increase by 2 after each execution. We could write it this way:

Example:

```
65  FOR M = 1 TO 99 STEP 2
```

The running variable, M, would be assigned a value of 1 during the first pass; a value of 3 during the second pass; 5 during the third pass and so on, until the value of M was 99 during the 50th (final) pass.

The running variable does not have to be a positive integer value; it can be a negative or decimal value. Also, the running variable can be made to decrease with each execution of the loop. This is done by specifying a negative value in the STEP clause. The initial, final, and STEP values assigned to the running variable can be expressed as variables or expressions as well as numbers.

Examples:

```
20   FOR A = .5 TO 1.5 STEP .1
30   FOR B = C TO 0 STEP -1
40   FOR D = F1 TO F2 STEP F3
50   FOR E = G/10 TO (A+B)**3 STEP L+1
```

Some important points to know and remember when creating a FOR-TO-NEXT loop are:

- The loop begins with a FOR-TO statement and ends with a NEXT statement.
- The same running variable name must be used in the FOR-TO and NEXT statements.
- The running variable can appear in a statement inside the loop, but its value cannot be changed.
- The running variable will be incremented by 1 unless otherwise specified by a STEP clause.
- If the initial and final values of the running variable are equal, and the step size is nonzero, the loop will be executed once.
- There are three conditions under which a loop will not be executed at all.
  1. The initial and final values of the running variable are equal and the step size is zero.
  2. The final value of the running variable is less than the original value, and the step size is positive.
  3. The final value of the running variable is greater than the original value, and the step size is negative.
- Control can be transferred out of a loop but not in. (The transfer out can be done by using a GOTO, an ON-GOTO, or an IF-THEN statement.)

Examine the following loop and see how it conforms to the points just listed.

Example:

```
600   FOR A = 0 TO 1.6 STEP .2
        .
        .
        .
650   LET X = A + B
660   IF X > X1 THEN 900
        .
        .
        .
700   NEXT A
        .
        .
        .
900   PRINT A,B,X
```

This example shows the use of the running variable (A) within the loop (line number 650). The statement in line number 660 will cause control to be transferred outside the loop, if the value of X is greater than the value of X1. Also, the running variable (A) that appears in the NEXT statement is the same as the running variable in the FOR-TO statement. These must be the same or the loop won't work. The step clause specifies that A is to be increased by .2 each time the loop is executed. If X does not exceed X1 the loop will be executed 9 times.

The following mortgage amortization program contains an example of the FOR-NEXT loop structure.

Example:

```
10    REMARK MONTHLY MORTGAGE AMORTIZATION
20    PRINT "ENTER MONTHLY PAYMENT (D) LOAN AMOUNT (B)";
30    PRINT "INTEREST RATE (I) NUMBER OF MONTHS (N)."
40    INPUT D,B,I,N
50    PRINT "MONTH";" PAYMENT";" LOAN BALANCE","PRINCIPLE",
60    PRINT "INTEREST"
70    LET R = I/12
80    FOR M = 1 to N
90    LET A = B*R
100   LET P = D - A
110   LET B = B - P
120   PRINT M;D;B,P,A
130   NEXT M
140   PRINT
150   PRINT "WITH ONE FINAL PAYMENT OF";B
999   END
```

The loop in this example is comprised of lines 80 through 130. Line 80 sets the initial value of M to 1 for the first execution of the loop. Lines 90 through 120 perform the calculations and print the results. Once the PRINT statement in line 120 has been executed, the NEXT statement in line 130 directs the computer to start the loop all over again. The loop will continue until the number of times it has been executed is equal to N (line 80). N is the number of months of the loan.

This example shows the use of a single loop structure using the FOR-TO. . .NEXT statements. It is also possible to have a loop within a loop. These are called *nested loops*.

Nested Loops

In addition to the rules which apply to single loops, the following rules apply to nested loops:

- Each nested FOR-NEXT loop must begin with its own FOR-TO statement and end with its own NEXT statement.
- An outer loop and an inner (nested) loop cannot have the same running variable.
- Each inner (nested) loop must be completely embedded within an outer loop, the loops cannot overlap.
- Control can be transferred from an inner loop to a statement in an outer loop or to a statement outside of the entire nest. However, control cannot be transferred to a statement within a nest from a point outside the nest.

The following example shows the structure of a nested loop.

Example:

```
65     FOR X = 1 TO 10
         .
         .
         .
90     FOR Y = X TO Z
         .
         .
         .
115    NEXT Y
         .
         .
         .
140    NEXT X
```

The inner loop (statements 90 through 115) is completely embedded within the outer loop (statements 65 through 140). Each loop begins and ends with its own FOR-TO and NEXT statements, and each loop has its own running variable. You will notice the running variable of the outer loop (X) is used as the initial value for the running variable of the inner loop (Y). This is allowed since the value of X is not changed within the inner loop.

By using nested loops, you are able to perform repeated sets of instructions within another set of instructions.

Example:

```
10    REM THIS PROGRAM WILL COMPUTE THE SQUARE, CUBE,
20    REM AND 4TH POWER OF THE NUMBERS 1 TO 10
30    FOR X = 1 TO 10
40    PRINT X,
50    LET A = X
60    FOR Y = 1 TO 3
70    LET A = A*X
80    PRINT A,
90    NEXT Y
100   PRINT " "
110   NEXT X
120   END
RUN
 1            1            1            1
 2            4            8            16
 3            9            27           81
 4            16           64           256
 5            25           125          625
 6            36           216          1296
 7            49           343          2401
 8            64           512          4096
 9            81           729          6561
 10           100          1000         10000
```

The outer loop (lines 30-110) will be executed 10 times, while the inner loop (lines 60-90) will be executed 3 times for each time the outer loop is executed. This means the inner loop is executed a total of 30 times.

WORKING WITH ARRAYS

In BASIC, we have the capability to store and reference data elements in lists or tables. These are called *arrays*. An entire array is assigned one name (e.g., D), and yet, we can refer to any element in the array by using a subscripted variable. If D is such an array, then D(1), read "D sub one,"

is the first element in array D. The value in parentheses is called a subscript. It indicates the relative position of a given element in an array. For example, LET D(3) = D(1) + D(2) is a BASIC statement which adds the first two numbers of array D and puts the sum in the third element of array D.

Before elements in arrays can be used, a method is needed to define the array. The **DIM** (DIMENSION) statement is used for this. It names the array and reserves memory space. For example, DIM D(15) would reserve space for 15 data elements with the name, D.

When subscripted variables are used, their corresponding values must be read and stored in memory during program execution. Examine the following example (one-dimensional array) and see how the data is stored in the computer's memory when the program statements are executed.

Example:

```
          .
          .
          .
750   FOR L = 1 TO 5
760   READ S(L)
770   NEXT L
780   DATA 72,74,69,70,71
          .
          .
          .
999   END
```

S(1)	S(2)	S(3)	S(4)	S(5)
72	74	69	70	71

In this example, the loop will be executed five times. The variable used to control the loop is also used as the subscript for S in the READ statement. On the first pass through the loop, the subscript L is 1, the value of L during the first execution of the loop. On the second pass, the subscript will be 2, and so on, until the value of L is equal to 5.

Subscripted variables can also be used to identify elements in tables (two-dimensional arrays) but it takes two subscripts, one to specify the row and a second one to specify the column. For example, S(2,5) would specify the location of the value in the second row, fifth column. Figure 5-1 is an example of a two-dimensional array.

DIM (DIMENSION) Statement

As stated earlier a method is needed to specify the size of an array. The BASIC programming language automatically assigns 11 elements to every

one-dimensional array and 121 elements (11 rows and 11 columns) to every two-dimensional array appearing in a program.

Larger arrays may be used. However, the size of each must be defined; that is, you must specify the maximum number of elements in each. The following example shows how a DIM statement is constructed.

Example:

```
10 DIM A(25), B(50,5)
```

This DIM statement would reserve memory space for an array named A with 25 elements, and an array named B with 50 rows and 5 columns. On some computers the first element in a one-dimensional array is referenced with subscript 0, and in a two-dimensional array by subscripts, 0, 0. If that is the case on your computer, you would set the upper limits at one less than the number of elements you need.

When the BASIC interpreter encounters a DIM statement such as the one above, it reserves an area in memory for arrays A and B made up of 25 and 250 elements respectively.

If an array requires less storage space than is automatically reserved by the BASIC interpreter it need not be defined by a DIM statement. However, by using a DIM statement only the space actually needed will be reserved.

Arrays

Some important things to remember about arrays:

- DIMENSION statements are used to define arrays.
- The elements in an array can be either numeric quantities or strings. However, all of the elements in a given array must be the same type (all numeric or all string).
- An array that contains numeric elements must be named with a single letter.
- A string array is referred to with a letter followed by a dollar sign.
- Elements in a one-dimensional array are referenced by the array name followed by a subscript in parentheses, A(1).
- Elements in a two-dimensional array, *matrix*, are referenced by the name followed by two subscripts in parentheses; the first references the row, the second the column, A(2,3).
- Subscripts may be a numeric-constant or a numeric-variable.
- Each array name in a program must be unique. However, an array and an ordinary variable can have the same name. Duplicating array names and variable names could be logically confusing; therefore, it is not a recommended practice.

Many computers have a special set of instructions called **MAT** instructions for working with matrices. By using a single MAT instruction, matrices may be defined, added, subtracted, multiplied, read, and manipulated in a variety of ways. Any of these operations may be done with FOR-NEXT loops; however, the MAT statement, if available, makes it easier. See Appendix II for examples and refer to the user's manual for your specific computer.

Sample Problem Using Nested Loops and a Matrix

Suppose you wanted to write a program to compute your career sea pay based on your paygrade and years of sea duty. You would need a two-dimensional array (matrix) to store the data. Figure 5-1 shows the table of values needed to determine sea pay.

Examine the following program and see how the matrix is constructed. The program contains nested loops (lines 20-60) which are used to read the values into the matrix. The outer loop sets up the row portion, G, which represents paygrade. The inner loop sets up the column portion, S, which represents the years of sea duty.

ARRAY P
(TWO-DIMENSIONAL ARRAY)

ROW (G) PAYGRADE		COLUMN (S) YEARS SEA DUTY 1	2	3	4	5	6	7	8	9	10	11	12
E-4	1	60	125	160	175	175	175	175	175	175	175	175	175
E-5	2	70	140	175	185	190	205	220	220	220	220	220	220
E-6	3	135	170	190	210	215	225	235	245	255	255	255	255
E-7	4	145	215	235	255	260	265	265	270	275	280	300	310
E-8	5	180	225	255	265	270	280	285	290	300	310	310	310
E-9	6	195	235	265	280	290	310	310	310	310	310	310	310

Figure 5-1.—Sea-pay table.

Example:

```
10    DIM P(6,12)
20    FOR G = 1 TO 6
30    FOR S = 1 TO 12
40    READ P(G,S)
50    NEXT S
60    NEXT G
70    PRINT "SEA PAY CALCULATION PROGRAM (E4-E9)"
80    PRINT "INPUT WHOLE NUMBERS ONLY"
90    PRINT "WHAT IS YOUR PAYGRADE"
100   INPUT G
110   LET G = G - 3
120   PRINT "HOW MANY YEARS SEA DUTY (1-12)"
130   INPUT S
140   PRINT "YOUR SEA PAY SHOULD BE $";P(G,S)
150   DATA 60,125,160,175,175,175,175,175,175,175,175,175
160   DATA 70,140,175,185,190,205,220,220,220,220,220,220
170   DATA 135,170,190,210,215,225,235,245,255,255,255,255
180   DATA 145,215,235,255,260,265,265,270,275,280,300,310
190   DATA 180,225,255,265,270,280,285,290,300,310,310,310
200   DATA 195,235,265,280,290,310,310,310,310,310,310,310
210   END

RUN
SEA PAY CALCULATION PROGRAM (E4-E9)
INPUT WHOLE NUMBERS ONLY
WHAT IS YOUR PAYGRADE
?8
HOW MANY YEARS SEA DUTY (1-12)
?7
YOUR SEA PAY SHOULD BE $  285
```

As seen in the output from this program, an E-8 with 7 years sea duty would receive $285.00 sea pay. Try the program and see what your sea pay would be.

The paygrade (4-9) is entered (line 100). Before it can be used as a subscript to determine row number, we subtract 3 (line 110). This makes it correspond to row number 1-6. Next, years of sea duty (1-12) are entered to be used as the subscript for column number. Then the PRINT statement (line 140) prints the corresponding value of the coordinates G and S from the matrix named P.

USING PREDEFINED FUNCTIONS

Some of the more commonly used mathematical functions have been predefined in the BASIC language. They were presented in Chapter 2 and are listed in Appendix I. To use them, all you have to do is specify the function and provide an argument (the number or variable, on which the function is to be executed).

Examples:

Calculate and Print the Square Root of 16

```
20  PRINT SQR (16)
```

Calculate the Absolute Value of X−Y

```
30  LET Z = ABS(X-Y)
```

If X = 10 and Y = 4, then X − Y = 6 and the absolute value is 6.

If X = 4 and Y = 10, then X − Y = −6 and the absolute value is also 6.

DEFINING YOUR OWN FUNCTIONS

In addition to the predefined functions, BASIC allows you to define your own functions within a program. The statement for defining functions is the **DEF** (DEFINE) statement.

The DEF statement consists of a statement number, the keyword DEF and the function definition. The function definition consists of the function name, followed by an equal sign, followed by a constant, variable, or expression. If the function requires an argument, then it must appear immediately after the function name, enclosed in parentheses. The following example shows how a function to convert Fahrenheit to Celsius could be defined.

Example:

```
40  DEF FNC(F) = (F-32)*5/9
```

Both numeric and string functions may be defined with the DEF statement. Numeric functions return numeric values and string functions return string values. Numeric function names must consist of three letters, the first two must be **FN**, followed by any single letter of the alphabet (A-Z). Therefore,

as many as 26 separate numeric functions can be defined in a single program (FNA,FNB,...FNZ).

String functions must consist of three letters followed by a dollar sign. Like numeric functions, the first two letters must be FN. Up to 26 separate string functions may be defined in a single program (FNA$,FNB$,...FNZ$). Numeric and string functions having the same three letters (FNA and FNA$) are considered as two different functions and may appear in the same program.

Some important things to remember about user defined functions are:

- A function definition statement must have a lower numbered line than that of the first reference to the function.
- The expression in a DEF statement is evaluated only when the defined function is referenced.
- If the execution of a program reaches a line containing a DEF statement, it proceeds to the next line with no other effect.
- A function definition can reference other defined functions, but not itself.
- A function may be defined only once in a program.
- Predefined functions may be used in arguments of user defined functions.
- Subscripted variables are not permitted as arguments in a function definition.

The following example shows a user defined function to calculate the area of a circle.

Example:

```
10  DEF FNA(R) = 3.1416*R**2
```

You can define your own functions, include them, and use them in your program. The following program shows the use of this function in a program to compute the areas of any number of circles.

Example:

```
10   DEF FNA(R) = 3.1416*R**2
20   PRINT "THIS PROGRAM WILL GIVE THE AREAS OF ANY NUMBER"
30   PRINT "OF CIRCLES THAT YOU SPECIFY"
40   PRINT "HOW MANY CIRCLES DO YOU WANT?"
50   INPUT N
60   FOR X = 1 TO N
70   PRINT "ENTER RADIUS"
80   INPUT Y
90   PRINT "CIRCLE # ";X,"RADIUS = ";Y, "AREA = ";FNA(Y)
100  NEXT X
110  END

RUN
```

```
THIS PROGRAM WILL GIVE THE AREA OF ANY NUMBER OF CIRCLES
THAT YOU SPECIFY
HOW MANY CIRCLES DO YOU WANT?
?5
ENTER RADIUS
?6
CIRCLE # 1    RADIUS = 6  AREA = 113.0976
ENTER RADIUS
?3
CIRCLE # 2    RADIUS = 3  AREA = 28.2744
ENTER RADIUS
?9
CIRCLE # 3    RADIUS = 9  AREA = 254.4696
ENTER RADIUS
?1
CIRCLE # 4    RADIUS = 1  AREA = 3.1416
ENTER RADIUS
?7
CIRCLE # 5    RADIUS = 7  AREA = 153.9384
```

The function to compute the area of a circle is defined in line 10. The PRINT statement, line 90, references the defined function to print the area of the circle. The variable name (R) used in the DEF statement is not the same as the one (Y) used in the PRINT statement where the function is referenced, rather it corresponds to the variable name used in the INPUT statement, line 80.

CONSTRUCTING AND USING SUBROUTINES

Like functions, subroutines are designed so they can be used over and over within a program, or so they can be inserted easily into other programs.

A *subroutine* is defined as a small program within another program. It does not have to be given a name or begin with a particular keyword. Subroutines can be used when sets of instructions are to be performed several

times in one or more programs. They can also be useful when more than one programmer is working on a program. Each programmer can be assigned a portion of the program to write, and that portion can be written as a subroutine. When all portions of the program have been written, they can be put together and referenced as subroutines in the main program. This reduces the possibility of statements written by one programmer conflicting with the statements written by another programmer.

To execute a subroutine, you must transfer control to the subroutine by using the keyword, **GOSUB**. Once executed, the subroutine transfers control back to the statement immediately following the GOSUB statement.

GOSUB and RETURN Statements

The **GOSUB** statement is used to transfer control to a subroutine. It is made up of a statement number followed by the keyword GOSUB and the number of the first statement in the subroutine.

The **RETURN** statement is used to transfer control back to the main program. It must be the last statement in the subroutine. The RETURN statement consists of a statement number followed by the keyword RETURN. When the RETURN statement is executed, it transfers control back to the main program to the statement immediately following the GOSUB statement. The structure is as follows:

```
30    GOSUB 200
40    .
      .
      .
200   LET X = Y + Z
      .
      .
      .
270   RETURN
```

Line 30 transfers control to line 200, the first statement of the subroutine. Line 270, the last statement of the subroutine, returns control to line 40, the line immediately following the GOSUB.

The following example shows the use of the GOSUB and RETURN statements in transferring control to a subroutine and returning control back to the main program.

Example:

```
10    REM THIS PROGRAM COMPUTES INTEREST
20    REM EARNED ON SAVINGS
30    PRINT "ENTER AMOUNT SAVED 1000-4999"
40    INPUT X
50    ON X/1000 GOTO 100,120,140,140
51    PRINT "VALUES ENTERED OUT OF RANGE, TRY AGAIN"
52    GOTO 30
60    PRINT "ANNUAL INTEREST ON $ ";X;"IS $ ";I
70    GOSUB 160
80    IF X$ = "Y" THEN 30
90    STOP
100   LET I = .04*X
110   GOTO 60
120   LET I = .045*X
130   GOTO 60
140   LET I = .05*X
150   GOTO 60
155   REM SUBROUTINE TO ASK IF MORE DATA
160   PRINT "ENTER Y IF MORE DATA"
170   PRINT "ENTER N IF NO MORE DATA"
180   INPUT X$
190   RETURN
999   END

RUN
ENTER AMOUNT SAVED
?1999
ANNUAL INTEREST ON $    1999 IS $    79.96
ENTER Y IF MORE DATA
ENTER N IF NO MORE DATA
?Y
ENTER AMOUNT SAVED
?2000
ANNUAL INTEREST ON $    2000 is $    90
ENTER Y IF MORE DATA
ENTER N IF NO MORE DATA
?N
```

The GOSUB statement in line 70 transfers control to the subroutine, line 160, which prints a prompt to the user. At this point, the subroutine gives the user the option, either to continue running the program or to terminate it. After a string value is entered at line 180, the RETURN statement, line 190 returns control to the main program at line 80. This line tests the value of the string variable, X$. If the value equals Y, control is transferred to line 30; if the value is equal to N, the program will STOP.

STOP Statement

The **STOP** statement, line 90, terminates execution of the program. Although the STOP statement terminates execution of the program, it does not replace the END statement. Remember, the END statement has two functions: to terminate execution of the program, and to indicate there are no more instructions for the BASIC interpreter to translate. Therefore you must include an END statement in every program, regardless of how many STOP statements you use. STOP statements may appear anywhere you need them in a program, and you can use as many as you want.

Summary

Constructing loops is made easier by using the FOR-NEXT loop structure. A FOR-NEXT loop always begins with a **FOR-TO** statement and ends with a **NEXT** statement. A numeric variable, called a running variable is used to control the number of times a loop is executed. The running variable used in the NEXT statement must be the same as the one used in the FOR-TO statement. The value of the running variable is incremented by one each time the loop is executed, unless a **STEP** clause is used to alter this. Control can be transferred out of a loop but not into one.

A loop may have another loop inside it. This is called a *nested loop*. Each nested loop must be completely embedded within the outer loop. They cannot overlap. Each nested loop must begin with its own FOR-TO statement and end with its own NEXT statement. Control cannot be transferred into a nested loop from a point outside the nest.

When working with *one- or two-dimensional arrays* you may reference any element in them by using a subscripted variable. The **DIM** (DIMENSION) statement is used to define the size of an array. It reserves space in the computer's memory for the specified number of elements.

Each array must be assigned a unique name, using either a numeric-variable name or a string-variable name. An array may contain either numeric or string data; however, all the elements in a given array must be of the same type (all numeric or all string).

Some of the more common mathematical functions have been predefined by the BASIC language; however, there are times when you may need to define your own. This can be done by using the **DEF** (DEFINE) statement.

A *subroutine* is a small program inside a larger program. It is useful when sets of instructions are to be performed several times in one or more programs, or when several programmers are working on the same program. A subroutine is executed by transferring control to the subroutine using a **GOSUB** statement with the statement number of the 1st statement in the subroutine. It is terminated by the keyword **RETURN**. When a RETURN

statement is executed, control is transferred from the subroutine back to the main program to the statement immediately following the GOSUB statement.

Execution of a program may be terminated anywhere in a program by the use of **STOP** statements.

CHAPTER 5

EXERCISES

1. Using a FOR-NEXT loop, write a program to print all the even numbers beginning with 20 and ending with 40.

2. Write a program that will compute the amount accumulated if you start with a penny and double it every day for 30 days. Print the total for each day.

3. Write a program that will read the following names into an array, then list them in reverse order. Carol, Chuck, Fred, Jane, John.

4. Write a program, including a DIM statement, to construct a matrix for 9 golfers with 4 games each. Include the capability to select and print a specific golfer number and a specific game. The scores for the players are:

	Game Number			
Golfer Number	1	2	3	4
1	69	72	70	75
2	73	72	74	70
3	71	75	69	73
4	70	74	72	71
5	69	68	70	72
6	75	77	73	70
7	68	66	70	72
8	70	73	71	69
9	76	71	74	72

5. Write a program containing a user defined function to compute the sale price for any piece of merchandise when given the original price; use 20% as the rate of discount.

6. Write a program containing a subroutine to calculate average miles per gallon.

CHAPTER 5

EXERCISE SOLUTIONS

The following programs present possible solutions to the exercises.

1.
```
10  FOR X = 20 TO 40 STEP 2
20  PRINT X
30  NEXT X
40  END

RUN

 20
 22
 24
 26
 28
 30
 32
 34
 36
 38
 40
```

2.
```
10   LET Y = .01
20   FOR X = 1 TO 30
30   LET Y =Y*2
35   PRINT Y
40   NEXT X
50   END

RUN

 2.00000000E-02
 4.00000000E-02
 8.00000000E-02
 .16
 .32
 .64
 1.28
 2.56
 5.12
 10.24
 20.48
 40.96
 81.92
 163.84
 327.68
 655.36
 1310.72
 2621.44
 5242.88
 10485.76
 20971.52
 41943.04
 83886.08
 167772.16
 335544.32
 671088.64
 1342177.28
 2684354.56
 5368709.12
 10737418.24
```

NOTE: Small numbers are represented by E notation.

3.
```
5   DIM A$(5)
10  DATA "CAROL"
20  DATA "CHUCK"
30  DATA "FRED"
40  DATA "JANE"
50  DATA "JOHN"
60  FOR X = 1 TO 5
70  READ A$(X)
80  NEXT X
90  FOR Y = 5 TO 1 STEP -1
100 PRINT A$(Y)
110 NEXT Y
120 END

RUN

JOHN
JANE
FRED
CHUCK
CAROL
```

4.
```
10  DIM G(9,4)
20  FOR P = 1 TO 9
30  FOR S = 1 TO 4
40  READ G(P,S)
50  NEXT S
60  NEXT P
```

```
70  PRINT "TOURNAMENT GOLF SCORES"
80  PRINT "ENTER PLAYER NUMBER (1-9)"
90  INPUT P
100 PRINT "ENTER GAME NUMBER (1-4)"
110 INPUT S
120 PRINT " PLAYER # ";P;" SCORE FOR GAME # ";S;"IS";G(P,S)
130 DATA 69,72,70,75,73,72,74,70,71,75,69,73
140 DATA 70,74,72,71,69,68,70,72,75,77,73,70
150 DATA 68,66,70,72,70,73,71,69,76,71,74,72
999 END

RUN

TOURNAMENT GOLF SCORES
ENTER PLAYER NUMBER (1-9)
?6
ENTER GAME NUMBER (1-4)
?2
PLAYER #6 SCORE FOR GAME #2 IS 77
```

5.
```
10  DEF FNP(C) = C-(.20*C)
20  PRINT "ENTER ORIGINAL PRICE"
30  INPUT Z
35  IF Z = 0 THEN 60
40  PRINT "SALE PRICE IS $ ";FNP(Z)
50  GOTO 20
60  END

RUN

ENTER ORIGINAL PRICE
?15
SALE PRICE IS $   12
```

6.
```
10  PRINT "ENTER MILES AND GALLONS"
20  INPUT M,G
30  GOSUB 60
40  PRINT "AVERAGE MILES PER GALLON IS ";A
50  STOP
60  LET A = M/G
70  RETURN
99  END

RUN

ENTER MILES AND GALLONS
?250,10
AVERAGE MILES PER GALLON IS      25
```

CHAPTER 6

FORMATTING PRINTED OUTPUT

PRINT, TAB, PRINT USING, and Format statement

In Chapters 3 through 5, we have explored the use of statements to get data into a program either using the READ and DATA statements or the INPUT statement. We have seen how to assign values to constants and variables, print the results of calculations, create and control loops, construct arrays, define and use functions, and write subroutines. However, our flexibility in printing output has been limited to the spacing automatically generated by a PRINT statement in BASIC. In this chapter, additional features generally available for printing output will be presented. These are the TAB function and the PRINT USING statement.

TAB FUNCTION

The **TAB** function allows output to be printed in any location of an output line. You are not limited to the standard spacing given by a comma, or the packed spacing given by a semicolon in a PRINT statement.

When a TAB function is used in a PRINT statement it must immediately precede the constant or variable to which it applies. The expression or number in parenthesis specifies the print position in which printing is to begin.

Example:

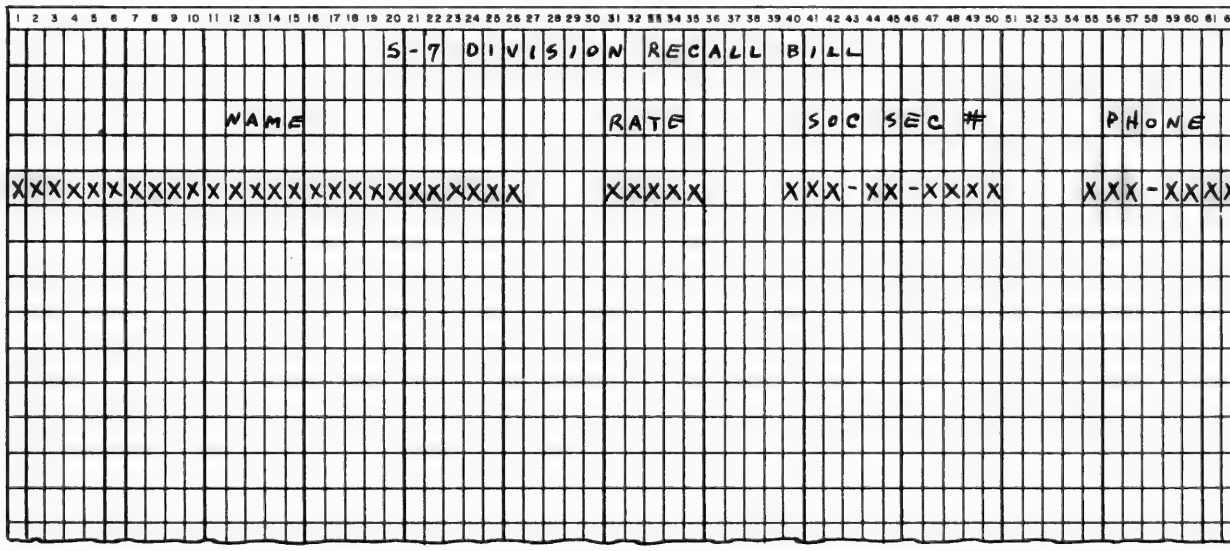
```
50 PRINT TAB(10);C$;TAB(30);A;TAB(40);D$
```

This PRINT statement with TAB functions causes the value represented by the string variable C$ to begin printing in print position 10, the value of the numeric variable A to begin printing in print position 30, and the value of the string variable D$ to begin printing in print position 40.

Care must be taken in specifying print positions with the TAB function because once a print position is passed, the printer does not backspace. It goes to the specified position on the next print line producing undesired results. For this reason, semicolons rather than commas are generally used as separators in PRINT statements with TAB functions. Remember, semicolons ignore print zones while commas are used for standard spacing (printing in zones). Therefore, it is recommended you use semicolons as separators when using the TAB function in PRINT statements.

The TAB function is useful in formatting various types of printed output. One use is in formatting headings for reports. The following example shows how this could be done.

Example:



In this example the report heading is to be centered on the page with the column headings spaced under it. By using a form like this you can get an idea of what the output will look like. Then when you code the TAB functions, use the position numbers to indicate where printing is to begin.

```
60  PRINT TAB(20);"S-7 DIVISION RECALL BILL"

70  PRINT

80  PRINT TAB(12);"NAME";TAB(31);"RATE";

90  PRINT TAB(41);"SOC SEC #";TAB(56);"PHONE"
```

In lines 60 through 90 the PRINT statements with the TAB functions are used to format the report and column headings. Using the TAB function enables you to print these headings anywhere you want them on the output page. The output would look like this:

S-7 DIVISION RECALL BILL

NAME RATE SOC SEC # PHONE

The form width for this example was sixty-two (62) positions. A formula that can be used to aid you in centering printed output is:

$$\frac{\text{Number of Print Positions} - \text{Number of Characters in Heading}}{2}$$

This will give you the number of blank print positions to be left on each side of the heading. If it works out that you have an unequal number of positions before and after the heading, the extra position can be put after the heading. However, this is left to your discretion.

In the example, the heading S-7 DIVISION RECALL BILL had 24 characters. Using the formula, we can determine the number of spaces to the left of the heading; (62 − 24)/2 = 19. Therefore, the first print position will be 20; this number is then used in parentheses following the TAB function.

Example:

```
60 PRINT TAB(20);"S-7 DIVISION RECALL BILL"
```

The column headings can be formatted in any way you want them. First determine how many print positions you want to allow for each data field, then determine how many characters the column headings will have. If you want the column headings centered over the data fields, use the same formula as for centering headings on the page, except substitute the number of print positions in a given field for total print positions. For example, the NAME field has 26 positions and NAME has four characters, thus (26 − 4)/2 = 11. Therefore, we start printing NAME in print position 12. For the remaining fields, you first determine how many spaces you want to leave between fields; this will give you the beginning position of each field. Now repeat the formula for determining the position of the column headings.

Let's examine a program which includes the TAB function to format the output. We'll use the mortgage amortization program. For the output, we want to print headings for MONTH, PAYMENT, LOAN BALANCE, PRINCIPAL, and INTEREST. Then we want to print the data in columns under the headings. A program to do this could be written this way:

Example:

```
10  PRINT "ENTER PAYMENT,LOAN AMOUNT,INTEREST RATE,# MONTHS"
20  INPUT D,B,I,N
30  PRINT TAB(1);"MONTH";TAB(8);"PAYMENT";TAB(22);"LOAN BAL";
40  PRINT "ANCE";TAB(41);"PRINCIPAL";TAB(58);"INTEREST"
50  PRINT
60  LET R = I/12
70  FOR M = 1 TO N
80  LET A = B*R
90  LET P = D-A
100 LET B = B-P
110 PRINT TAB(3);M;TAB(8);D;TAB(20);"$";B;TAB(38);"$";P;
120 PRINT TAB(55);"$";A
130 NEXT M
140 PRINT
150 PRINT TAB(1);"WITH ONE FINAL PAYMENT OF";TAB(27);"$";B
900 END
RUN
ENTER PAYMENT,LOAN AMOUNT,INTEREST RATE,# MONTHS
? 379.16,36000,.12,36
```

MONTH	PAYMENT	LOAN BALANCE	PRINCIPAL	INTEREST
1	379.16	$ 35980.84	$ 19.16	$ 360
2	379.16	$ 35961.4884	$ 19.3516	$ 359.8084
3	379.16	$ 35941.943284	$ 19.545116	$ 359.614884
4	379.16	$ 35922.20271684	$ 19.74056716	$ 359.41943284
5	379.16	$ 35902.26474401	$ 19.9379728316	$ 359.2220271684
6	379.16	$ 35882.12739146	$ 20.1373525599	$ 359.0226474401
7	379.16	$ 35861.78866538	$ 20.3387260854	$ 358.8212739146
8	379.16	$ 35841.24655204	$ 20.5421133462	$ 358.6178866538
9	379.16	$ 35820.49901757	$ 20.7475344796	$ 358.4124655204
10	379.16	$ 35799.54400775	$ 20.9550098243	$ 358.2049901757
11	379.16	$ 35778.37944783	$ 21.1645599225	$ 357.9954400775
12	379.16	$ 35757.00324231	$ 21.3762055217	$ 357.7837944783
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
36	379.16	$ 35174.64701082	$ 27.1421088038	$ 352.0178911962

WITH ONE FINAL PAYMENT OF $ 35174.64701082

In this example the TAB functions caused the computer to tabulate to the specified print positions and print whatever values followed each TAB function. In the PRINT statements, lines 30 and 40, the values were the column headings; in lines 110 and 120, dollar signs followed by the values of the variables were printed, beginning in the specified print positions. The space between the dollar sign and the number is reserved for the minus sign, if the number is negative.

In the previous example, the headings look nice, but look at the numbers. You would not want an amortization schedule with cents expressed to five or more decimal places, and without commas to make the dollar values easier to read. Although we used the TAB function to specify where the column should be printed, we let the computer format each output field automatically. By using the PRINT USING statement we can override the print-zone restrictions and further enhance the appearance of the output.

PRINT USING STATEMENT

The **PRINT USING** statement allows greater flexibility in formatting printed output. You can "dress up" your output. You can define the format you want with a format statement, then use a PRINT USING statement to print the output in accordance with the specifications in the format line. The

general structure of the PRINT USING statement and format line is:

```
30   PRINT USING 100,Y
40   PRINT USING 110
      .
      .
      .
100  %     $##.##
110  %     MORTGAGE AMORTIZATION SCHEDULE
```

The PRINT USING statement, line 30, specifies line 100 is to be used as the FORMAT line for printing variable Y. Line 100 specifies the number of positions to be printed and the format in which they are to be printed. For example, if Y = 25.782, it would print as $25.78. If it were 05.3753, it would print as $5.37.

The PRINT USING statement, line 40, specifies line 110 is to be used as its format line. Line 110 is a different type of format line. It specifies where the literal, MORTGAGE AMORTIZATION SCHEDULE, is to be printed. Literals are words or phrases in a format statement that are to be printed exactly as they appear in the format statement.

The per cent sign used as the first character in lines 100 and 110 indicates that these are format lines. Some computers use a colon; some use the letters FMT. Refer to your user's reference manual for applicable characters.

Format control characters within a format line are used to describe the output images desired, and to control spacing and positioning of data in the output line. The types of control characters and their syntax may vary depending on the computer you are using. However, they usually include #, $, comma(,), and decimal point(.).

- # — specifies positions for alphanumeric and numeric data.
- $ — is used with financial data.
- , — is used in numeric data to provide clarity.
- . — is used to specify the placement of a decimal point in the printed line.

When specifying an output image, be sure to allow enough positions for the largest possible number or character string in that field. If you fail to allow enough positions in a given field, either truncation of some of the data will occur; or, you will get an error message indicating the field is too small. You should pay close attention to this possibility and ensure it doesn't happen.

Format lines may contain a combination of literals and output images. However, if a format line only specifies literals (i.e., headings), it will not contain format control characters.

The PRINT USING statement includes the line number of the format statement followed by any variables or expressions to be printed. The items in the PRINT USING statement are separated by commas. However, if you are printing only literals, the PRINT USING statement will contain only the line number of the format statement.

Now that we have seen the mechanics of the PRINT USING statement, let's use it to modify the mortgage amortization program to print the output in a more realistic format.

```
10    PRINT "ENTER PAYMENT,LOAN AMOUNT,INTEREST RATE,# MONTHS"
20    INPUT D,B,I,N
30    PRINT USING 40
40    %MONTH    PAYMENT    LOAN BALANCE    PRINCIPAL    INTEREST
50    PRINT
60    LET R = I/12
70    FOR M = 1 TO N
80    LET A = B*R
90    LET P = D-A
100   LET B = B-P
110   PRINT USING 120,M,D,B+.005,P+.005,A+.005
120   %###        $#,###.##      $###,###.##        $#,###.##      $#,###.##
130   NEXT M
140   PRINT
150   PRINT USING 160,B+.005
160   %WITH ONE FINAL PAYMENT OF $###,###.##
900   END
```

Examine the PRINT USING statements in lines 30, 110, and 150. Each of these statements refers to another line number, which is the format statement where the format of the output is described. The PRINT USING statement in line 30 causes the information in line 40 to be printed exactly as it appears.

The PRINT USING statement, line 110, causes the values for the variables listed to be printed according to the images specified in line 120. In format line 120, we have reserved three spaces, ###, for month, which is represented by the first variable name, M, in line 110. Next $#,###.## reserves space for the payment (variable D). Then $###,###.## reserves space for the loan balance. The next two images are for principal and interest and are the same structure as the image for payment. Note the use of the dollar sign, comma, and decimal point in the images for payment, loan balance, principal and interest; these

will appear in the output as appropriate. Also, note that .005 was added to variables B, P, and A in order to round them.

Now examine the PRINT USING statement in line 150 and its associated format line, line 160. Line 160 contains a literal and an image specification. The literal will be printed exactly as it appears in the format line, and the rounded value for variable B will be printed in the specified positions including the dollar sign, comma, and decimal point as appropriate.

The output from the modified program would look like this:

MONTH	PAYMENT	LOAN BALANCE	PRINCIPAL	INTEREST
1	$379.16	$35,980.84	$19.16	$360.00
2	$379.16	$35,961.49	$19.35	$359.81
3	$379.16	$35,941.94	$19.55	$359.61
4	$379.16	$35,922.20	$19.74	$359.42
5	$379.16	$35,902.26	$19.94	$359.22
6	$379.16	$35,882.13	$20.14	$359.02
7	$379.16	$35,861.79	$20.34	$358.82
8	$379.16	$35,841.25	$20.54	$358.62
9	$379.16	$35,820.50	$20.75	$358.41
10	$379.16	$35,799.54	$20.96	$358.20
11	$379.16	$35,778.38	$21.16	$358.00
12	$379.16	$35,757.00	$21.38	$357.78
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
36	$379.16	$35,174.65	$27.14	$352.02

WITH ONE FINAL PAYMENT OF $35,174.65

SUMMARY

Both the **TAB** function in a **PRINT** statement and the **PRINT USING** statement allow you greater flexibility in formatting your printed output than the spacing automatically generated by a PRINT statement.

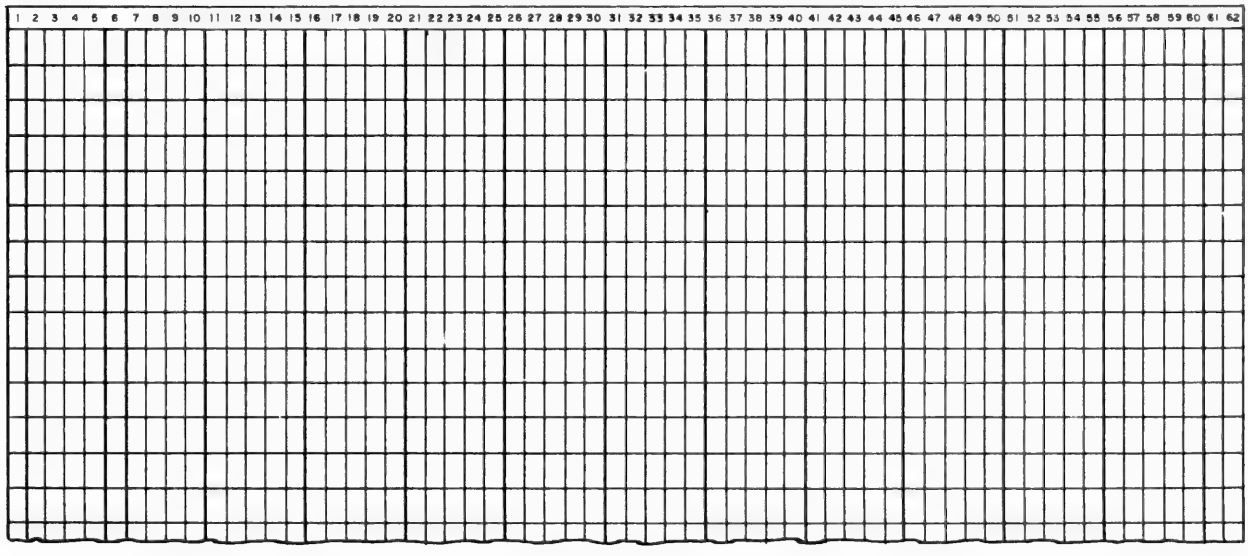
With a TAB function you can specify exactly where you want an item to be printed in an output line. It causes the computer to tabulate to a specified print position. The semicolon rather than the comma is generally used as a separator in PRINT statements containing TAB functions.

The other way to enhance printed output is with the PRINT USING statement. It allows even more flexibility than the TAB function. With it, you can print literals or print the values of variables (or expressions) in a given format, specified in a format line. It allows you to line up your output so that it is more meaningful and easier to read. Literals specified in a format line are printed exactly as they appear. The values for variables are printed in the location and image specified in a format line. Format control characters are used to describe output images and to position data in the output line. When specifying images, be sure to allow enough spaces for the largest possible number or character string in that field.

CHAPTER 6

EXERCISES

1. Layout and print three columns of data with headings. The headings are: NAME, RANK, and SERVICE NUMBER. The data is: name (allow 26 positions), rank (allow 5 positions), and service number (allow 11 positions). Assume a report width of 60 positions. Write PRINT statements using TAB to center the headings over each column of data and print the data. As nearly as possible, allow an equal number of spaces between columns.



2. Assume the numeric-variable D can have a range of values from .0001 to 99999.99. Write a PRINT USING statement and associated format line to format and print the numeric-variable D. Include dollar sign, comma, and decimal point. Use the rounding method shown in the chapter to round off to two decimal places.

3. Remember the program from Chapter 5, which doubled the value of a penny every day for 30 days. It is listed below. Look at the

output; the numbers are rather lengthy and awkward to read. Modify the program to print the values with:

A. the decimal points lined up
B. a $ at the beginning of each value
C. commas in appropriate positions

```
10   LET Y = .01
20   FOR X = 1 TO 30
30   LET Y = Y*2
35   PRINT Y
40   NEXT X
50   END
```

```
RUN
 2.00000000E-02
 4.00000000E-02
 8.00000000E-02
 .16
 .32
 .64
 1.28
 2.56
 5.12
 10.24
 20.48
 40.96
 81.92
 163.84
 327.68
 655.36
 1310.72
 2621.44
 5242.88
 10485.76
 20971.52
 41943.04
 83886.08
 167772.16
 335544.32
 671088.64
 1342177.28
 2684354.56
 5368709.12
 10737418.24
```

4. Write a program to compute and print the discounted prices of a list of items for any discount rate. The discount rate should be input at the time the program is executed. Put a title and column headings on the printed report. Allow 10 positions for item name and up to $9,999.99 for unit price.

   Use the following list of items as the data:

Items	Unit Price
TIRES	$ 379.95
BATTERY	61.95
ENGINE	1032.50
FUSES	.55

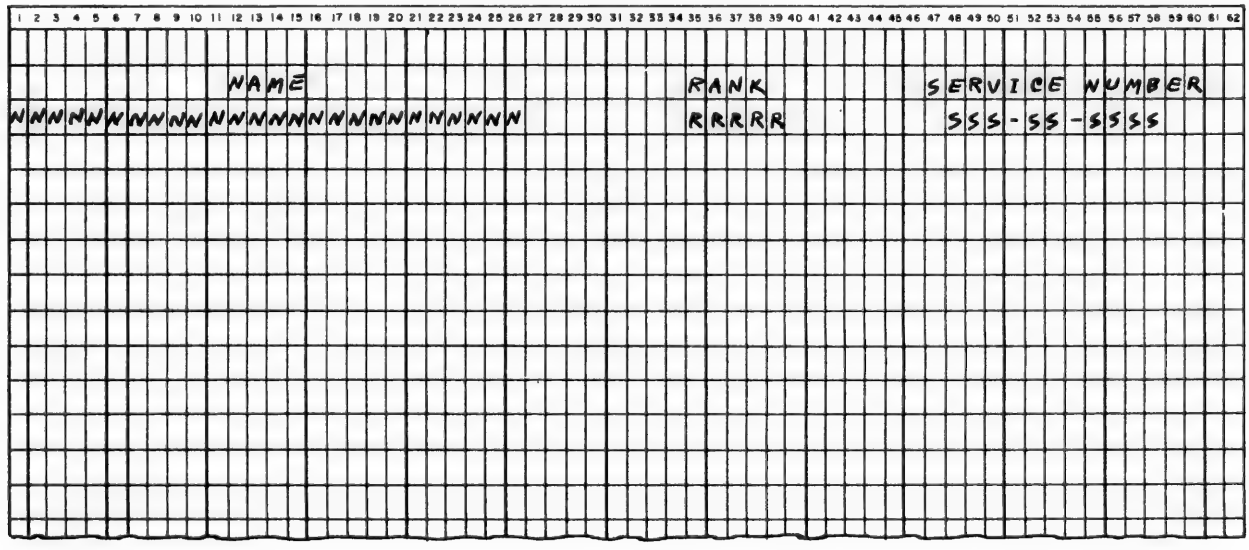
CHAPTER 6

EXERCISE SOLUTIONS

The solutions to the exercises could be written this way:

1. 25 PRINT TAB(12);"NAME";TAB(35);"RANK";TAB(47);"SERVICE NUMBER"

   35 PRINT TAB(1);N$;TAB(35);R$;TAB(48)S$



Since the service number heading is longer than the data, your spacing may be one (1) or two (2) spaces different from the one shown, depending on how you choose to arrange the odd number of spaces.

2. 40 PRINT USING 50,D+.005

   50% $##,###.##

   Positioning of the output data in the output line is left to your discretion. The primary concern is that you know how to construct the image statement and perform the rounding function.

3.
```
10   LET Y = .01
20   FOR X = 1 TO 30
30   LET Y = Y*2
35   PRINT USING 36,Y
36   %$###,###,###.##
40   NEXT X
99   END

RUN
```

```
            $0.02
            $0.04
            $0.08
            $0.16
            $0.32
            $0.64
            $1.28
            $2.56
            $5.12
           $10.24
           $20.48
           $40.96
           $81.92
          $163.84
          $327.68
          $655.36
        $1,310.72
        $2,621.44
        $5,242.88
       $10,485.76
       $20,971.52
       $41,943.04
       $83,886.08
      $167,772.16
      $335,544.32
      $671,088.64
    $1,342,177.28
    $2,684,354.56
    $5,368,709.12
   $10,737,418.24
```

4.
```
10    REM THIS PROGRAM GIVES THE DISCOUNTED
11    REM PRICES FOR A LIST OF ITEMS
15    DIM I$(4),U(4)
20    PRINT "ENTER DISCOUNT RATE,.XX"
30    INPUT D
40    PRINT USING 50,D*100
50    %DISCOUNTED PRICES AT ##%
60    PRINT USING 70
70    %  ITEM                    PRICE
80    FOR X = 1 TO 4
90    READ I$(X),U(X)
100   LET P = U(X) - (U(X)*D)
110   PRINT USING 120,I$(X),P + .005
120   %##########          $#,###.##
130   NEXT X
200   DATA "TIRES",379.95,"BATTERY",61.95
210   DATA "ENGINE",1032.50,"FUSES",.55
900   END
RUN
ENTER DISCOUNT RATE,.XX
?.10
DISCOUNTED PRICES AT 10%
ITEM            PRICE
TIRES           $341.96
BATTERY          $55.76
ENGINE          $929.25
FUSES             $0.50
```

CHAPTER 7

STORING AND RETRIEVING PROGRAMS AND DATA

Up to this point, we have dealt with reading data, using READ and DATA statements, inputting data, using the INPUT statement, and printing output. We have not discussed storing and retrieving programs or storing and retrieving data. It seems useless to write a program, key it in, get it to work, and then be unable to save it to run again later without keying it in again. As you write programs to solve more complex problems with large amounts of data, you'll need ways to save your programs and data on some type of storage medium. You will probably use a magnetic disk or tape.

The way this is done varies considerably from one computer to another. Therefore, we will not attempt to present all the possible ways. The primary concern is that you are aware that there are ways to store and retrieve both programs and data. Your computer user's reference manual will provide you with detailed instructions.

STORING AND RETRIEVING PROGRAMS

First, let's examine one method for storing a program. When storing a program, you must have a way to reference it in order to load it to be run again. To do this, you assign it a name. Program names are a series of characters you choose. Depending on the computer you are using, you will probably be limited to six or eight characters. There may also be limitations on specific characters that can be used. Most computers require the first character in a program name to be alphabetic.

If you are using a computer which has more than one programming language available, you will have to specify the programming language you wish to use. The following example shows how this might be done.

Example:

```
SYSTEM              BASIC
OLD OR NEW          NEW
NEW FILE NAME       MPG
READY
```

In the first line of the example, the computer asks what programming language you want to use by displaying the word, SYSTEM. You enter BASIC; then it asks if you want to retrieve an OLD program or enter a NEW one. You enter NEW. The computer then asks for the NEW FILE NAME; you enter MPG as the program name. The computer then responds with the message, READY. This indicates the computer is ready for you to key in your program. You key in your program in the same manner as you have throughout this course.

```
10    LET M1 = 0
20    LET G1 = 0
30    DATA 485,16.5,450,14,418,11,432,12.5
40    FOR X = 1 TO 4
50    READ M,G
60    LET M1 = M1 + M
70    LET G1 = G1 + G
80    NEXT X
90    LET A = M1/G1
100   PRINT "TOTAL MILES ARE";M1;"TOTAL GALLONS ARE";G1
110   PRINT "AVERAGE MILES PER GALLON IS";A
999   END
SAVE
FILE MPG SAVED
```

Once you have entered the program, you key in the system command, **SAVE**, which instructs the computer to store your program either on disk or tape. The computer then responds with the message, FILE MPG SAVED.

To recall a saved program, you respond with OLD to the computer query NEW or OLD. The computer will then ask for the program name. After you have entered the program name, the computer loads the program into memory. It is then available to be executed, changed, or listed.

To SAVE a program on some computers, all that is required is to key in the program, enter the system command, SAVE, followed by the program name enclosed in quotation marks.

Example:

```
SAVE "MCASH"
```

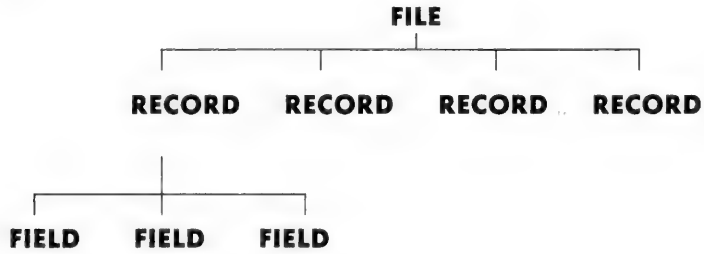
This command will save the program.

STORING AND RETRIEVING DATA

The programs we have written thus far dealt with data either stored in the program with DATA statements or input via an INPUT statement at time of program execution. These statements may be sufficient when handling data that is to be processed only once. However, there are times when you will want to save your data to be used by other programs. You are restricted in what you can do with the data when it is stored in DATA statements in a program. It is impractical to update data or to use the same data with different programs. It would be very time consuming to go through all the DATA statements and make changes, or to enter all new DATA statements each time you wanted to update and/or process the data. To eliminate having to enter the data each time it is to be processed, it can be stored independently on an auxiliary storage medium such as disk or tape. This keeps the data separate from the program so that it can be used with different programs without being rekeyed. Data stored in this manner is called a data file.

A *data file* is any group of related records, such as inventory, personnel, payroll, manhour accounting, and so on. A *record* is composed of *data fields* which are specified areas of a record used for a particular category of data. For example, in a parts inventory file you would have a record for each part stocked. Each part and all the associated data about that part (part number, part name, and unit price) would make up a record. Part number, part name, and unit price are each a field. The following example shows a pictorial breakdown of a file composed of records and associated fields.

Example:



You might say, "All of this is fine, but how do I create and use these files?" Since the BASIC file processing statements are not standardized, you will have to refer to the user's reference manual for your computer to find the file processing statements and their syntax. We will attempt only to deal with the file processing statements conceptually.

Solving Problems With File Processing Statements

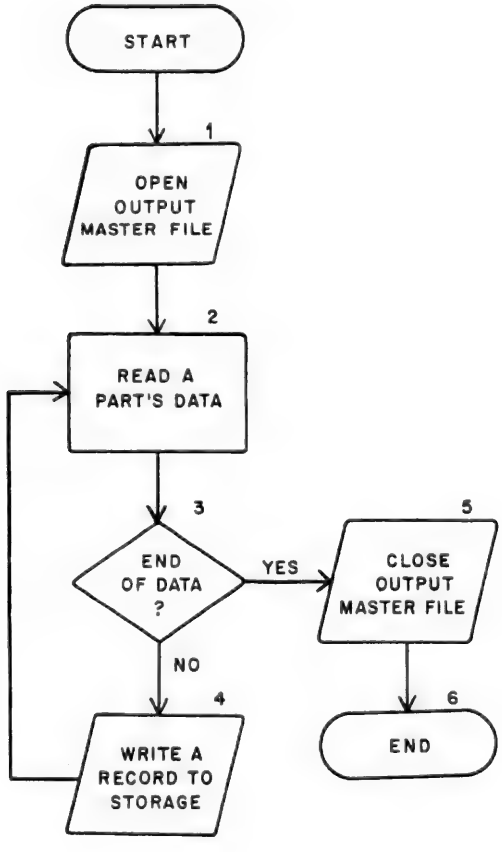
Suppose you were keeping track of an inventory file for an auto parts store. You would need to create a file which would contain a record for each part in the inventory. This type of file is called a master file. Each record will contain the part number, part name, on-hand quantity, unit price, and reorder

point (stock level). The flowchart in figure 7-1 describes one procedure that could be used to create the master file.

Before you can write data records to a storage medium, you usually have to set up the file. Most computers do this with some form of *open* statement that names the file and makes it available for processing (block 1). Once the file is available for processing, the program will read the data elements from DATA statements just as in the past (block 2). Then the program checks for end-of-data (block 3); if there is data, it will write a record to the output file for each part in the inventory (block 4). Once all the data has been read and written you must tell the computer you are finished with that file. This is usually done with some form of *close* statement (block 5). At this point the computer does any necessary end-of-file processing then terminates the program (block 6).

With a fundamental understanding of the problem and a flowchart as a guide, we are ready to write a program to create the master file. The program could be written as shown in figure 7-1.

The **DATA SAVE** statement in line 10 opens the output file, making it available for the program to write data out to it, and assigns it the file name, INVFILE. The #2 tells the computer what storage medium is being used. Lines




```
05   REM THIS PROGRAM CREATES A MASTER FILE
10   DATA SAVE #2,OPEN "INVFILE"
20   READ P,N$,Q,U,R
30   IF N$ ="ENDDATA" THEN 60
40   DATA SAVE #2,P,N$,Q,U,R
50   GOTO 20
60   DATA SAVE #2,END
200  DATA 0517,"SOCKET",8,2.95,5
210  DATA 7340,"SHACKLE",10,12.48,10
220  DATA 1034,"SPARKPLUGS",25,16.42,15
230  DATA 3054,"TIRES",9,64.50,4
240  DATA 9121,"BRAKE SHOES",20,15.45,26
250  DATA 5230,"BATTERY",16,49.00,8
260  DATA 6120,"POINTS",50,6.40,20
270  DATA 5510,"CONDENSER",12,3.62,20
500  DATA 0,"ENDDATA",0,0,0
999  END
```

Figure 7-1.—Flowchart and program to create a master file.

20 through 50 form a loop which reads data from DATA statements and writes records to the master file until all the data has been read and written. In line 20, the READ statement reads the data from the DATA statements, and line 30 tests the data for the "ENDDATA" record which indicates end-of-data. If it is not end-of-data, then the DATA SAVE statement in line 40 writes a record containing the values for the variable names listed. It writes the record onto the master file (INVFILE). The GOTO statement in line 50 transfers control back to line 20 to repeat the reading, testing, and writing of records until the READ statement reads the "ENDDATA" record and the IF-THEN test proves true. Then the program transfers control to line 60 which writes an end-of-file character on the master file and closes the file. The next executable statement is 999 which terminates execution of the program.

For simplicity we have included only the data for one record in each DATA statement. This makes it easier to visualize what a record looks like. It also makes it easier to correct any data that is keyed incorrectly. Notice also that the DATA statements begin with much higher line numbers. This separates them from the program steps and allows room for expanding the program or for the addition of data without having to renumber the program. Using "ENDDATA" to indicate end-of-data rather than using a FOR-NEXT loop allows more flexibility in handling various amounts of data and eliminates having to know how much data will be processed.

In this example, the DATA SAVE statements are used to open a file and write a file name on it (line 10), write data to the output file (line 40), and write an end-of-file indicator on the file (line 60). Since the file processing statements will probably be different on the system you are using, you will have to refer to the user's reference manual to determine what statements perform these functions.

Once the master file has been created you might want to print it, either to ensure that all items have been entered and entered correctly; or you may wish to have a listing on which you can keep a tally of items sold and use it when you update the master file. The following program could be used to print the master file.

Example:

```
10  REM THIS PROGRAM PRINTS MASTER FILE INVFILE
20  DATA LOAD #2,"INVFILE"
30  PRINT "PART NUMBER","PART NAME","QUANTITY","PRICE",
40  PRINT "REORDER POINT"
50  DATA LOAD #2,P,N$,Q,U,R
60  IF END THEN 99
70  PRINT P,N$,Q,U,R
80  GOTO 50
99  END
```

In this program, the **DATA LOAD** statement, line 20, opens the master file (INVFILE) and makes it available for processing. The PRINT statements in lines 30 and 40 print the column headings. Lines 50 through 80 comprise the loop which will control the reading and printing of the master file. The DATA LOAD statement in line 50 reads the values for the variables and makes them available for the PRINT statement to print. The **IF END** statement in line 60 checks for end-of-file. At end-of-file, control is transferred to line 99 where the END statement terminates the program.

When reading a file and the end-of-file indicator is read, the computer will detect it. If any future attempt is made to read from the file before it is reopened, an end-of-file error message is printed and the computer stops executing the program. When reading a file, the BASIC statement IF END can be used to test whether the computer did, in fact, read the end-of-file indicator. It also transfers control to any specified line number in a program. For example, you could transfer control to a PRINT statement to print summary totals, or transfer control to an END statement to terminate the program as was done in this example. Depending on what computer you are using, the IF END statement is normally placed either immediately following or immediately preceding a read statement.

Now that we have created and printed the master file, we can now begin to use it. For example, we'll need to update the file so the on-hand quantity for each item will be current and we'll know when a particular item needs to be reordered. The flowchart (figure 7-2, a foldout at the end of this chapter) is a pictorial representation of the procedures required to solve the problem. The problem solution has been divided into four parts:

1. Read the part's data into arrays.
2. Update the quantity on hand.
3. Produce a list of items below stock level.
4. Create a new master file.

While some of these parts could have been combined, they were kept separate for ease of understanding.

Study the flowchart and examine its four distinct parts. Part one consists of blocks 1 through 4 which are used to read the parts data into arrays and to accumulate a total count of the records in the file. This total will be used to control successive FOR-NEXT loops.

Part two consists of blocks 5 through 12 which are the update portion. This section allows you to select part numbers at random to be updated. Using the part number entered in block 5, the FOR-NEXT loop compares it to each part number in array P. When a match is found, the subscript X equals the part's relative position in the arrays and control is transferred to block 11. Block 11 allows you to enter quantity sold. Block 12 uses the value of X as the subscript to select the corresponding on-hand quantity, Q(X), to be updated. Q(X) is used with the quantity sold, Q1, to compute the new on-hand quantity. Then control is transferred back to block 5 for another part number to be entered. If a match is not found, this means the entire array has been searched in the FOR-NEXT loop, blocks 7 through 9, and the part number (P1) entered in block 5 did not match any part number in the parts number array, P(X). Therefore, an error message is printed and control is transferred back to block 5 to enter the corrected part number

or another part number. If no more updating is required, 9999 is entered and block 6 tests this condition. If it proves true, control is transferred to block 13 which is the first block of part three.

Part three consists of blocks 13 through 17 which generate a list of parts below stock level. Blocks 14 through 17 comprise a FOR-NEXT loop. Block 15 tests if the on-hand quantity is greater than the reorder-point quantity. If this proves false, the part number, part name, and unit price of that part is printed on the below stock level list. Block 17 tests for X equal to K. If it is not equal, control is transferred to block 14 which increments the value of X by one and continues searching the records for items to be reordered. Once X is equal to K (the number of records in the file), control is transferred to block 18, which is the first block of part four.

Part four consists of blocks 18 through 22 which create the new master file. Blocks 19 through 21 comprise a FOR-NEXT loop which is used to control the writing of records to the new master file. Block 20 writes the new records onto an auxiliary storage medium; block 21 tests for X equal K. If X is not equal to K, control is transferred back to block 19 which increments X by 1 and block 20 writes another master record. Once X is equal to K, control is transferred to block 22 which closes the new master file and the program is terminated, block 23.

Examine the program (figure 7-2) and see how the program coding corresponds to the blocks in the flowchart.

In this program, lines 50 through 130 are used to read the master file into the computer's memory. Lines 150 through 300 are used to update the master file. In line 190 the part number to be updated is entered; or if no more parts are to be updated, 9999 is entered. Line 200 tests P1 for end of update data (9999); if not, control goes to line 210 which is the beginning of a FOR-NEXT loop that searches array P for a matching part number. If a matching part number is not found, control is then transferred to line 240 which prints the message, PART # NOT FOUND. If a matching part number is found, control is transferred to line 260 which displays the data elements of the record to be updated. Line 270 requests quantity sold to be entered. Line 290 computes new on-hand quantity by subtracting Q1 from Q(X). Then the GOTO statement in line 300 returns control to line 180 which is a prompt to enter another part number to be updated.

When no more parts records are to be updated, control is transferred to line 340 which prints the heading for the below stock level list. Lines 350 through 380 comprise a FOR-NEXT loop which is used to search the on-hand quantity and the reorder point quantity arrays. Line 360 tests the updated on-hand quantity, Q(X), and the reorder point quantity, R(X), to determine if Q(X) is greater than R(X). If it is, the data for another record is tested. If it is less than or equal to R(X), the part number, part name, and unit price are printed on the list. Once all parts have been tested for reorder criteria, lines 400 through 470 write the updated master file to an auxiliary storage medium.

In this program we chose to read all the data into arrays to allow updating the parts records in any sequence, and to allow a part's record to be updated more than once in a single run. This will work for small data files. However, for larger data files (those that cannot fit into memory) other processing methods would be needed.

SUMMARY

As you write programs to be used repetitively and to solve more complex problems, the need for storing and retrieving programs becomes more apparent. Although the specific statements for storing and retrieving programs will vary on different computers, the concepts will be similar. Programs must be assigned a name so they can be referenced and reloaded. A program name is a series of characters used to identify a program. A system command, such as SAVE, is used to store a program. To reload a program will require another system command in which you specify the name of the program to be loaded. Once a program is loaded into a computer's memory, it is available to be executed, changed, or listed.

The need for a method to store and retrieve data files becomes apparent when the same files are to be used extensively. When data is stored in a program, you are restricted in what can be done with it. This data cannot be used with other programs and it would be very difficult, if not impossible, to update. To eliminate these restrictions, data can be stored on auxiliary storage media such as magnetic disk or tape. Data stored in this manner is called a data file. A file consists of a group of related records and each record consists of fields. A field is a specified area in a record used for a particular category of data.

Most computers require a file processing statement to make a file ready to be read or written. This is usually some type of *open* statement and includes the file name and the tape or disk where the data will be written or read. Once the file is opened, file processing statements are used to read data from a storage medium into memory or to write data from memory to a storage medium. Once you are through processing a data file, you must tell the computer you are finished with that file. This is done with some type of *close* statement.

Storing data independently from programs allows more flexibility. Data stored in this manner can be used by different programs. It can be updated, printed, or used to generate various reports.

The programming concepts presented in this text are fundamental; they are intended to get you started in programming. As your knowledge and experience increase, you will discover new and more sophisticated ways to use the language to solve more difficult problems.

CHAPTER 7

EXERCISES

1. Draw a flowchart and write a program to create a file to keep track of stock purchases. Use the file processing statements presented in this chapter. The file should include a record for each stock purchased. Each record should be composed of three data fields: name of stock, purchase price per share, and number of shares bought. Use the INPUT statement to enter the data for the file.

2. Four candidates are running for public office. To win, a candidate must receive over 50% of the votes cast. Draw a flowchart and write a program to accumulate total votes for each candidate and determine if a candidate has won or if a runoff election is required. Print the result and print each candidate's name and total votes for each. The file to be used as input is stored on tape or disk and consists of records from six districts. Each record contains the following fields:

   1. Candidate number (1-4)—variable X

   2. Candidate name—string variable N$

   3. District number (1-6)—variable D

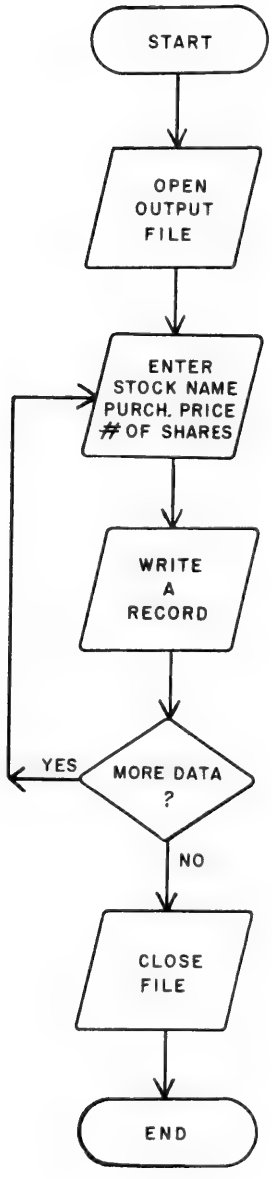
   4. Number of votes—variable V

   The records in the file are in no particular order.

CHAPTER 7
EXERCISE SOLUTIONS

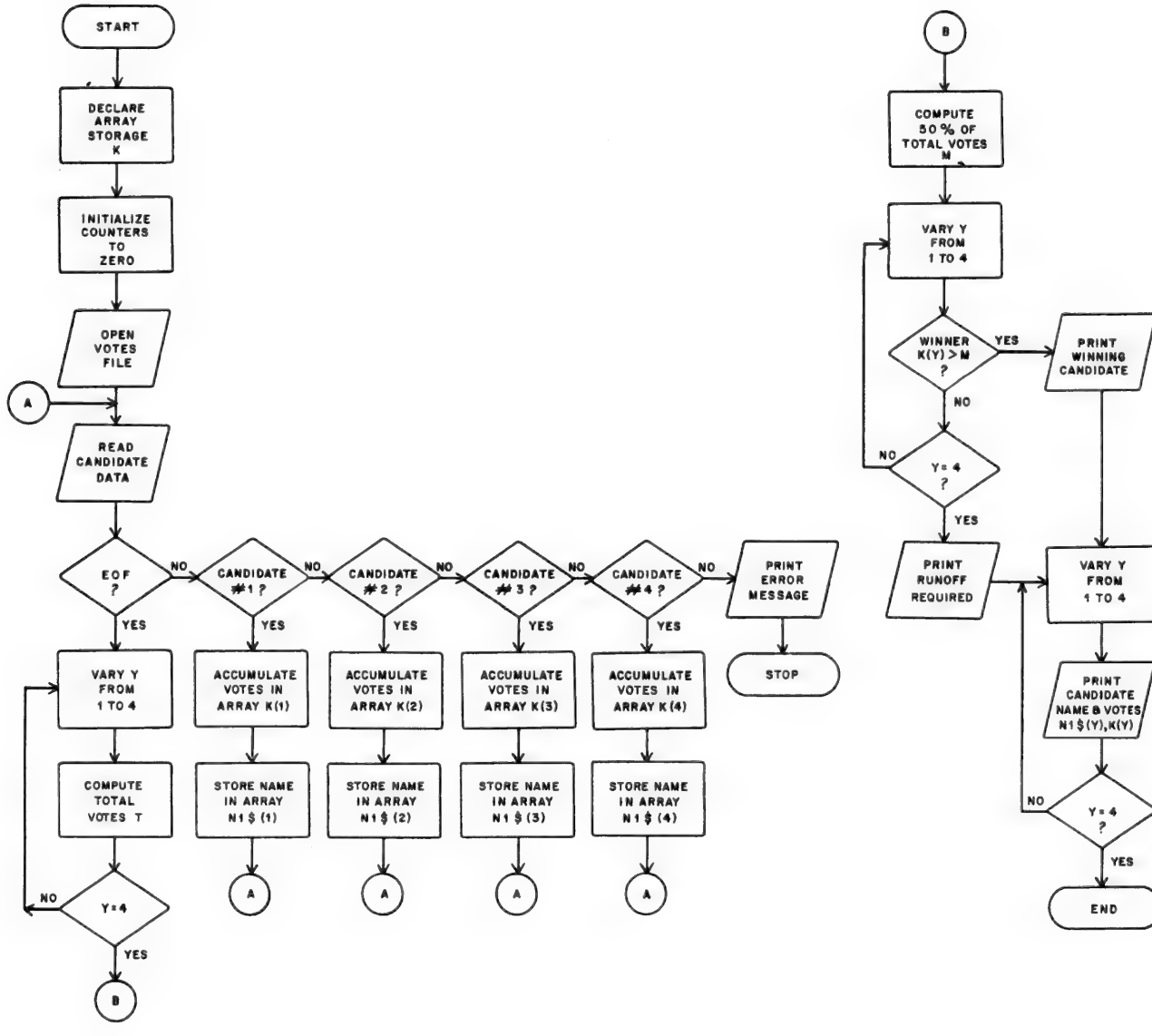
The following flowcharts and programs present possible solutions to the exercises.

1. Stock Purchase Flowchart and Program




```
05  REM STOCK PURCHASE PROGRAM
10  DATA SAVE #2,OPEN"STKFILE"
20  PRINT "ENTER STOCK NAME,PURCHASE PRICE,# OF SHARES"
30  PRINT "ENTER END,0,0 IF NO MORE DATA"
40  INPUT N$,P,S
50  IF N$ = "END" THEN 80
60  DATA SAVE #2,N$,P,S
70  GOTO 40
80  DATA SAVE #2,END
99  END
```

2. Election Results Flowchart and Program



```
05  REM ELECTION RESULTS PROGRAM
10  DIM K(4)
15  DIM N1$(4)
20  LET T = 0
30  FOR Y = 1 TO 4
40  LET K(Y) = 0
50  NEXT Y
60  DATA LOAD #2,"VOTES"
70  DATA LOAD #2,X,N$,D,V
80  IF END THEN 230
90  ON X GOTO 110,140,170,200
100 PRINT "ERROR IN CANDIDATE NUMBER, RUN TERMINATED"
105 STOP
110 LET K(1) = K(1) + V
120 LET N1$(1) = N$
130 GOTO 70
140 LET K(2) = K(2) + V
150 LET N1$(2) = N$
160 GOTO 70
170 LET K(3) = K(3) + V
180 LET N1$(3) = N$
190 GOTO 70
200 LET K(4) = K(4) + V
210 LET N1$(4) = N$
220 GOTO 70
230 FOR Y = 1 TO 4
240 LET T = T + K(Y)
250 NEXT Y
260 LET M = .5*T
270 FOR Y = 1 TO 4
280 IF K(Y) > M THEN 320
290 NEXT Y
300 PRINT "RUNOFF REQUIRED"
310 GOTO 330
320 PRINT N1$(Y),"WON!"
330 FOR Y = 1 TO 4
340 PRINT N1$(Y),K(Y)
350 NEXT Y
999 END
```

You may have noticed that the data field for candidate number contains a 1, 2, 3, 4, and that this code could be used as a subscript to reference the correct locations in the arrays. To modify the program to take advantage of this, you could replace statements 90 through 220 with the following statements:

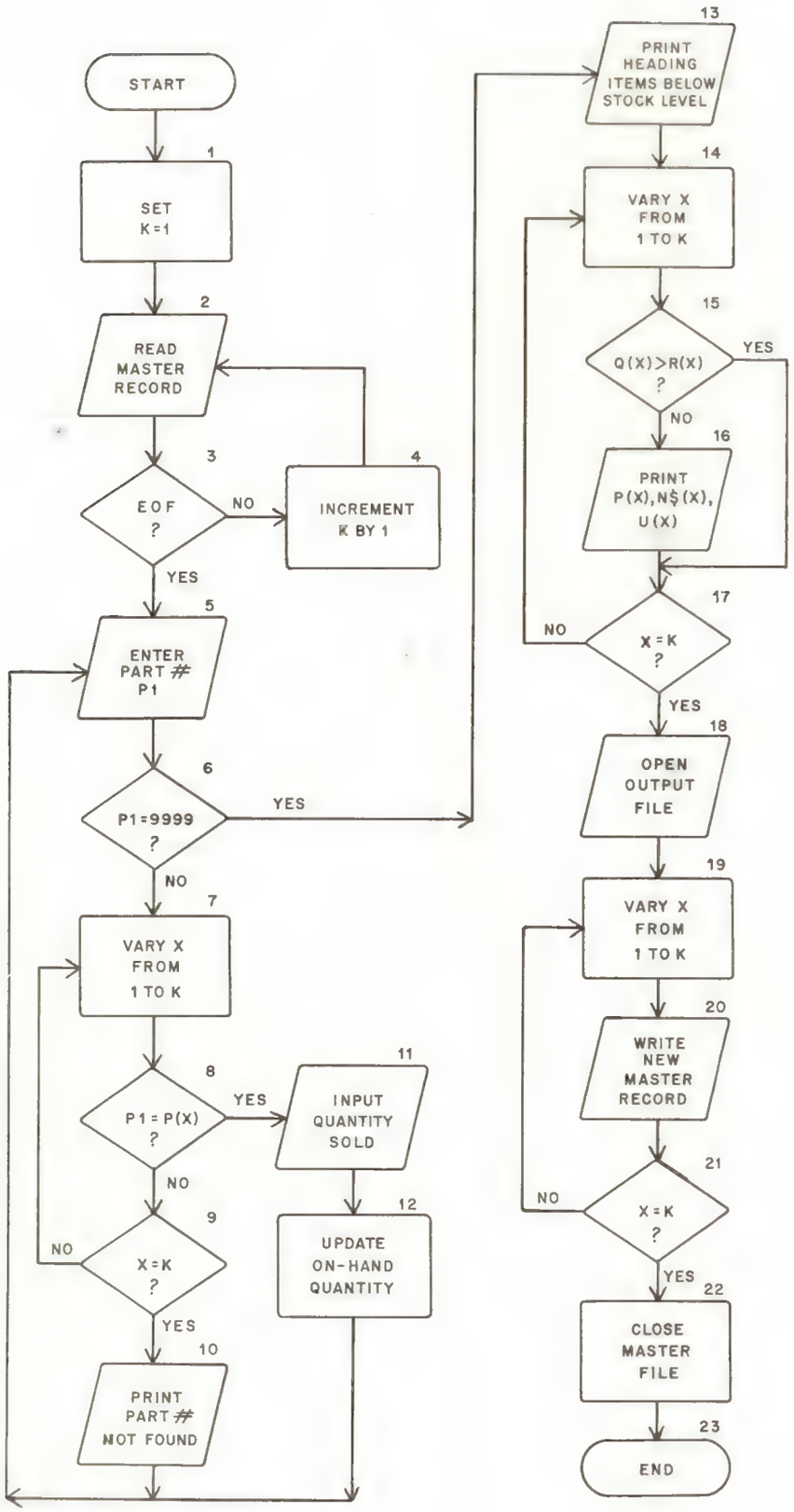
```
90    LET K(X) = K(X) + V

100   LET N1$(X) = N$

110   GOTO 70
```

When the value of candidate number is 1, the number of votes cast for candidate 1 would be added to K(1); when it's 2, the number of votes cast for candidate 2 would be added to K(2); and so on.

When you are defining data and data codes, it is a good idea to think about how you are going to use the data in the programs. You may be able to save yourself coding time.




```
10    REM THIS PROGRAM UPDATES THE PARTS MASTER FILE,
20    REM PRINTS A LIST OF PARTS BELOW STOCK LEVEL AND
30    REM CREATES A NEW MASTER FILE
40    REM
50    REM PART 1 (BLOCKS 1-4) LOAD THE MASTER FILE
60    REM
70    LET K = 1
80    DIM P(8),N$(8),Q(8),U(8),R(8)
90    DATA LOAD #2,"INVFILE"
100   DATA LOAD #2,P(K),N$(K),Q(K),U(K),R(K)
110   IF END THEN 170
120   LET K = K+1
130   GOTO 100
140   REM
150   REM PART 2 (BLOCKS 5-12) UPDATE THE ON-HAND QUANTITY
160   REM
170   LET K = K-1
180   PRINT "ENTER PART #,9999 IF NO MORE DATA"
190   INPUT P1
200   IF P1 = 9999 THEN 340
210   FOR X = 1 TO K
220   IF P1 = P(X) THEN 260
230   NEXT X
240   PRINT "PART # NOT FOUND"
250   GOTO 180
260   PRINT P(X),N$(X),Q(X),U(X),R(X)
270   PRINT "ENTER QUANTITY SOLD"
280   INPUT Q1
290   LET Q(X) = Q(X)-Q1
300   GOTO 180
310   REM
320   REM PART 3 (BLOCKS 13-17) PRINT THE BELOW STOCK LIST
330   REM
340   PRINT "THESE ITEMS ARE BELOW STOCK LEVEL"
350   FOR X = 1 TO K
360   IF Q(X)>R(X) THEN 380
370   PRINT P(X),N$(X),U(X)
380   NEXT X
390   REM
400   REM PART 4 (BLOCKS 18-21) WRITE THE UPDATED
410   REM MASTER FILE TO STORAGE
420   REM
430   DATA SAVE #3,OPEN "INVFILE"
440   FOR X = 1 TO K
450   DATA SAVE #3,P(X),N$(X),Q(X),U(X),R(X)
460   NEXT X
470   DATA SAVE #3,END
999   END
```


Figure 7-2.—Parts inventory flowchart and program.

APPENDIX I

SUMMARY OF BASIC KEYWORDS, VARIABLE NAMES, AND PREDEFINED FUNCTIONS

KEYWORD	DESCRIPTION	EXAMPLES	CHAPTER REFERENCE
DATA	Stores numeric or string data in the program	**DATA 4,5,6,7** **DATA "JOE","JIM"**	4
DEF	Defines a function	**DEF FNA(X) = X/4**	5
DIM	Defines one- and two-dimensional arrays	**DIM A$(20),B(5,10)**	5
END	Indicates to the interpreter/compiler there are no more instructions and terminates execution of the program	**END**	3
FOR/NEXT	**FOR** begins a **FOR-NEXT** loop and specifies the control of the loop	**FOR X = 1 TO Y STEP 2**	5
	STEP specifies the value to be used to increment the running variable		
	NEXT transfers control to the beginning of the **FOR-NEXT** loop	**NEXT X**	5
GOSUB/RETURN	**GOSUB** transfers control to a subroutine	**GOSUB 80**	5
	RETURN transfers control from a subroutine back to the statement immediately following the GOSUB statement	**RETURN**	5
GOTO	Unconditionally transfers control to the statement specified	**GOTO 120**	4
IF THEN	If the condition proves true control is transferred to the specified statement number	**IF X = 2 THEN 70**	4
INPUT	Allows user to enter data during program execution	**INPUT X,Y$**	4
LET	Assigns the value of a constant, variable, or an expression to a variable	**LET X = 1** **LET X = X + 1**	3

KEYWORD	DESCRIPTION	EXAMPLES	CHAPTER REFERENCE
ON GOTO	Selects the line number of the next statement to be executed based on its positional relationship to the integer value of the variable or expression, and transfers control to that statement	**ON X/10 GOTO 70,90,110**	5
PRINT	Prints the value of any constant, variable, or expression	**PRINT 10** **PRINT X** **PRINT (X+Y)/2** **PRINT "MESSAGE"**	3
PRINT USING	Used in conjunction with a format line to print output in accordance with specifications in format line	**100 PRINT USING 110,N$** **110%###############** **70 PRINT USING 80,U,S** **80% $#,###.## $###.##**	6
READ	Assigns variable names to data in **DATA** statements	**READ C(X,Y)** **READ A,B$**	4
REM	Allows for programmer comments in the program	**REM THIS PROGRAM** **REM COMPUTES MILES** **REM PER GALLON**	2
RESTORE	Resets the pointer to the top of the data list. Used after **READ** statements when data is to be read more than once.	**RESTORE**	4
STOP	Terminates execution of the program. Can be used anywhere in a program. Does not replace the **END** statement	**STOP**	5
TAB	Used in conjunction with **PRINT** statement to specify in what print position printing is to begin	**PRINT TAB(25);C$** **PRINT TAB(40);"RATE"**	6

APPENDIX II

MAT STATEMENTS

As stated in Chapter 5, the functions that can be performed by **MAT** (matrix) statements can be done with **FOR-NEXT** loops. However, if your computer has MAT statements, it is more convenient to use one MAT statement to create a matrix than it is to use nested FOR-NEXT loops to do the same thing.

A matrix containing numbers can be read, inputted, added, subtracted, multiplied, printed, or manipulated in various ways. However, before any matrix operations can be performed, the computer must know the size of the matrix. The size of the matrix is defined by using DIM statements at the beginning of the program, for example, **DIM P(6,12)**.

To illustrate the convenience and power of MAT operations, let's use the sea pay program from Chapter 5.

```
10   DIM P(6,12)
20   FOR G = 1 TO 6
30   FOR S = 1 TO 12
40   READ P(G,S)
50   NEXT S
60   NEXT G
70   PRINT "SEA PAY CALCULATION PROGRAM (E4-E9)"
80   PRINT "INPUT WHOLE NUMBERS ONLY"
90   PRINT "WHAT IS YOUR PAYGRADE"
100  INPUT G
110  LET G = G - 3
120  PRINT "HOW MANY YEARS SEA DUTY (1-12)"
130  INPUT S
140  PRINT "YOUR SEA PAY SHOULD BE $";P(G,S)
150  DATA 60,125,160,175,175,175,175,175,175,175,175,175
160  DATA 70,140,175,185,190,205,220,220,220,220,220,220
170  DATA 135,170,190,210,215,225,235,245,255,255,255,255
180  DATA 145,215,235,255,260,265,265, 270,275,280,300,310
190  DATA 180,225,255,265,270,280,285,290,300,310,310,310
200  DATA 195,235,265,280,290,310,310,310,310,310,310,310
210  END
```

Lines 20 through 60 can be replaced by one MAT statement:

```
20 MAT READ P
```

Remember, line 10 is still needed to define the size of the matrix. One **MAT READ** will read all the data into a matrix named "P," with 6 rows and 12 columns.

The elements in the matrix will still be referenced by the values input at lines 100 and 130. The method for identifying the elements in a matrix is the same as that used in Chapter 5. Each element in a matrix is identified by its row number and column number in a subscripted variable.

Suppose we wanted to print the entire matrix, we would include a statement such as:

```
65 MAT PRINT P
```

This one statement would cause the 72 elements in matrix P to be printed.

Another function that can be performed on a matrix is multiplication. Suppose there was a 4% increase in sea pay, and you wanted to update the sea pay table without rekeying the data. One way this could be accomplished is by multiplying all the elements in the matrix by 104. The MAT instruction to do this would look like this:

```
68 MAT P = (104)*P
```

This would cause all the elements in matrix P to be multiplied by 104 and the updated sea pay rates written back into the matrix named P. By multiplying the matrix by 1.04% (104), it eliminates computing the 4% increase and having to add the increase back into the matrix. The new rates are computed and automatically written back into the matrix named P.

The following list shows some MAT statements with an explanation of their function.

MAT Statement	Explanation of Function
MAT INPUT X	Allows user to input a matrix called X via the console (terminal)
MAT PRINT Z	Prints a matrix called Z
MAT READ M	Reads a matrix called M
MAT D = (N)*P	Defines matrix D as equal to the constant number N times matrix P (sea pay table example)
MAT C = X*Y	Defines matrix C as being equal to matrix X times matrix Y
MAT U = ZER	Completely fills matrix U with zeroes
MAT B = CON	Completely fills matrix B with ones

For more detailed instructions on the use of MAT instructions, consult your computer user's reference manual.

APPENDIX III

GLOSSARY

ARRAY—An arrangement of elements in a meaningful pattern of one or more dimensions.

CODING SHEETS—Forms or sheets of paper on which programs are written.

CONDITIONAL TRANSFER STATEMENT—A statement used to alter the normal sequence of program execution based on the result of the evaluation of a condition.

CONSTANT—An element whose value does not change during program execution.

DATA—Elements that represent information which may be processed or produced by a computer.

DECISION TABLE—A tabular method to show all the elements of a problem and what actions are to be taken based on all possible conditions of the elements. Used in problem analysis to document a problem solution.

EXPRESSION—A combination of symbols (constants, variables, arithmetic operators, and functions) used to define a desired computation or calculation.

FIELD—In a record, a specified area used for a particular category of data.

FILE—A collection of related records treated as a unit.

INCREMENT—To increase the value of a counter.

INTEGER—A positive or negative number without fractional parts.

INTERACTIVE—A computer environment that permits the user to respond directly to the actions of individual programs during their execution.

KEYWORDS—Words defined in the BASIC programming language and used in statements to identify the type of statement, such as LET, PRINT, READ, and DATA.

LINE NUMBER—A unique number placed at the beginning of each BASIC statement which serves as a statement label and program sequence number.

LOOP—A set of program instructions to be executed repetitively.

MACROINSTRUCTION—An instruction in a source language that is equivalent to a specified sequence of machine instructions.

MATRIX—A two-dimensional array.

MEMORY—An area in a computer which is capable of storing data and programs.

MNEMONIC SYMBOL—A symbol selected because it is easy to remember; e.g., "mpy" for "multiply."

PRINT ZONE—A specified area in a printed output line where the results of a print statement element may be printed. Each output line consists of multiple zones.

PROGRAM—A series of instructions in a given language which instructs a computer to perform specified operations to solve a problem.

PROGRAM EXECUTION—The interpretation and performance of the program instructions by the computer.

PROGRAMMING LANGUAGE—A language designed to express computer programs.

RECORD—A collection of related items of data treated as a unit.

SIGN—The arithmetic symbol used to identify a positive or negative value.

STATEMENT—A generalized source language instruction to a computer telling it to do a specified operation or series of operations.

SUBSCRIPT—A number, variable, or expression used with an array name to indicate the relative location of an element in a given array.

SYNTAX—The rules governing the structure of a language.

SYSTEM COMMAND—A command that can be given directly to the computer instructing it to do something with the system or a program, such as RUN, SAVE, and LIST.

TERMINAL—A human-to-machine and machine-to-human communication device that provides a means of interaction between an executing program and a person. Normally, a terminal is composed of a keyboard entry device and an interconnected display such as a printer or a cathode-ray tube.

UNCONDITIONAL TRANSFER STATEMENT—A statement used to alter the normal sequence of program execution regardless of any condition.

VARIABLE—An element whose value may change during program execution.

INDEX

G

I

K

L

M

N

O

P

R

S

*U.S. GOVERNMENT PRINTING OFFICE: 1985-631-087:20048

OFFICER-ENLISTED CORRESPONDENCE COURSE

INTRODUCTION TO PROGRAMMING IN BASIC

NAVEDTRA 10079-2

Congratulations! By enrolling in this course, you have demonstrated a desire to improve yourself and the Navy. Remember, however, this self-study course is only one part of the total Navy training program. Practical experience, schools, selected reading, and your desire to succeed are also necessary to successfully round out a fully meaningful training program. You have taken an important step in self-improvement. Keep up the good work.

HOW TO COMPLETE THIS COURSE SUCCESSFULLY

ERRATA: If an errata comes with this course, make all indicated changes or corrections before you start any assignment. Do not change or correct the textbook or assignments in any other way.

TEXTBOOK ASSIGNMENTS: The textbook for this course is Introduction to Programming in BASIC, NAVEDTRA 10079-2. The textbook pages that you are to study are listed at the beginning of each assignment. Study these pages carefully before attempting to answer the questions in the course. Pay close attention to tables and illustrations because they contain information that will help you understand the text. Read the learning objectives provided at the beginning of each chapter or topic in the text and/or preceding each set of questions in the course. Learning objectives state what you should be able to do after studying the material. Answering the questions correctly helps you accomplish the objectives.

SELECTING YOUR ANSWERS: After studying the text, you should be ready to answer the questions in the assignment. Read each question carefully, then select the BEST answer. Be sure to select your answer from the subject matter in the textbook. You may refer freely to the textbook and seek advice and information from others on problems that may arise in the course. However, the answers must be the result of your own work and decisions. You are prohibited from referring to or copying the answers of others and from giving answers to anyone else taking the same course. Failure to follow these rules can result in suspension from the course and disciplinary action by the Commander, Naval Military Personnel Command.

SUBMITTING COMPLETED ANSWER SHEETS: It is recommended that you complete all assignments as quickly as practicable to derive maximum benefit from the course. However, as a minimum, your schedule should provide for the completion of at least one assignment per month--a requirement established by the Chief of Naval Education and Training. Failure to meet this requirement could result in disenrollment from the course.

TYPES OF ANSWER SHEETS: If you received Automatic Data Processing (ADP) answer sheets with this course, the course is being administered by the Naval Education and Training Program Development Center (NAVEDTRAPRODEVCEN), and you should follow the instructions in paragraph A below. If you did NOT receive ADP answer sheets with this course, you should use the manually scored answer sheets attached at the end of the course and follow the directions contained in paragraph B below.

A. ADP Answer Sheets

All courses administered by the NAVEDTRAPRODEVCEN include one blank ADP answer sheet for each assignment. For proper computer processing, use only the original ADP answer sheets. Reproductions are not acceptable.

Recording Information on the ADP Answer Sheets: Follow the "MARKING INSTRUCTIONS" on the answer sheet. Be sure that blocks 1, 2, and 3 are filled in correctly. This information is necessary for your course to be properly processed and for you to receive credit for your work.

As you work the course, be sure to mark your answers in the course booklet because your answer sheets will not be returned to you. When you have completed an assignment,

transfer your answers from the course booklet to the answer sheet.

Mailing the Completed ADP Answer Sheets: As you complete each assignment, mail the completed ADP answer sheet to:

Commanding Officer
Naval Education and Training
Program Development Center
Pensacola, FL 32559-5000

The answer sheets must be mailed in envelopes, which you must either provide yourself or get from the local Educational Services Officer (ESO). You may enclose more than one answer sheet in a single envelope. Remember, regardless of how many answer sheets you submit at a time, the NAVEDTRAPRODEVCEN should receive at least one a month. NOTE: DO NOT USE THE COURSE COMMENTS PAGE AS AN ENVELOPE FOR RETURNING ANSWER SHEETS OR OTHER COURSE MATERIALS.

Grading: The NAVEDTRAPRODEVCEN will grade your answer sheets and notify you by letter of any incorrect answers. The passing score for each assignment is 3.4. Should you get less than 3.4 on any assignment, a blank ADP answer sheet will be enclosed with the letter listing the questions incorrectly answered. You will be required to redo the assignment and resubmit a new completed answer sheet. The maximum score that can be given for a resubmitted assignment is 3.4.

Course Completion: When you complete the last assignment, fill out the "Course Completion" form in the back of the course and enclose it with your last answer sheet. The NAVEDTRAPRODEVCEN will issue you a letter certifying that you satisfactorily completed the course. You should ensure that credit for the course is entered in your service record.

Student Questions: Any questions concerning this course should be referred to the NAVEDTRAPRODEVCEN by mail using the address listed above or by telephone: AUTOVON 922-1329, FTS 948-1329, or commercial (904) 452-1329.

B. Manually Scored Answer Sheets

If you did not receive ADP answer sheets with this course, it is being administered by your local command and you must use the answer sheets attached at the end of the course booklet.

Recording Information on the Manually Scored Answer Sheets: Fill in the appropriate blanks at the top of the answer sheet. This information is necessary for your course to be properly processed and for you to receive credit for your work. As you work the course, be sure to mark your answers in the course booklet, because your answer sheets will not be returned to you. When you have completed an assignment, transfer your answers from the course booklet to the answer sheet.

Submitting the Completed Manually Scored Answer Sheets: As you complete each assignment, submit the completed answer sheet to your ESO for grading. You may submit more than one answer sheet at a time. Remember, you must submit at least one assignment a month.

Grading: Your ESO will grade the answer sheets and notify you of any incorrect answers. The passing score for each assignment is 3.4. Should you get less than 3.4 on any assignment, the ESO will not only list the questions incorrectly answered but will also give you a pink answer sheet marked "RESUBMIT." You will be required to redo the assignment and complete the "RESUBMIT" answer sheet. The maximum score that can be given for a resubmitted assignment is 3.4.

Course Completion: After you have submitted all the answer sheets and have earned at least a 3.4 on each assignment, your command will give you credit for this course by making the appropriate entry in your service record.

Student Questions: Any questions concerning the administration of this course should be referred to your ESO.

NAVAL RESERVE RETIREMENT CREDIT

This course is evaluated at 3 Naval Reserve retirement points, which will be credited upon satisfactory completion of the two assignments.

Points	Assignments
3	2

These points are creditable to personnel eligible to receive them under current directives governing retirement of Naval Reserve personnel.

Naval reserve officers should anticipate some delay in delivery of course completion

certification as certifications must be forwarded via Naval Officer Record Support Activity for recording and endorsement.

COURSE OBJECTIVES

In completing this OCC-ECC, you will demonstrate a knowledge of the BASIC computer programming language by correctly answering questions. Specifically, you should be able to:

DIFFERENTIATE between the problem solving steps in programming. DESCRIBE the purpose and characteristics of flowcharts. RECOGNIZE flowcharting symbols and their meanings. DESCRIBE the final steps in the programming process.

DESCRIBE the fundamental concepts, structure, and syntax of BASIC.

DESCRIBE the uses of the END and PRINT statements in BASIC programs and RECOGNIZE their syntax. IDENTIFY the symbols used for arithmetic operations. WRITE BASIC statements to solve arithmetic expressions. DEFINE and RECOGNIZE constants and variables.

CODE statements to introduce data into a computer system. SELECT and USE transfer of control statements to alter the normal sequence of program execution and to control program loops.

DEFINE the structure and rules of FOR-NEXT loops. DEFINE and USE arrays. DESCRIBE functions and subroutines, including their syntax and keywords.

CODE statements to format printed output.

DESCRIBE files and file handling concepts, terminology, and processing.

Naval courses may include a variety of questions -- multiple-choice, true-false, matching, etc. The questions are not grouped by type; regardless of type, they are presented in the same general sequence as the textbook material upon which they are based. This presentation is designed to preserve continuity of thought, permitting step-by-step development of ideas. Some courses use many types of questions, others only a few. The student can readily identify the type of each question (and the action required) through inspection of the samples given below.

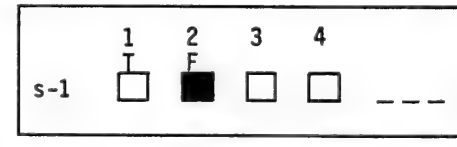
MULTIPLE-CHOICE QUESTIONS

Each question contains several alternatives, one of which provides the best answer to the question. Select the best alternative, and blacken the appropriate box on the answer sheet.

SAMPLE

s-1. The first person to be appointed Secretary of Defense under the National Security Act of 1947 was
1. George Marshall
2. James Forrestal
3. Chester Nimitz
4. William Halsey

Indicate in this way on the answer sheet:



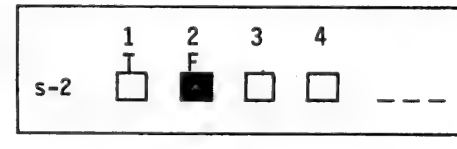
TRUE-FALSE QUESTIONS

Mark each statement true or false as indicated below. If any part of the statement is false the statement is to be considered false. Make the decision, and blacken the appropriate box on the answer sheet.

SAMPLE

s-2. Any naval officer is authorized to correspond officially with any systems command of the Department of the Navy without his commanding officer's endorsement.

Indicate in this way on the answer sheet:



MATCHING QUESTIONS

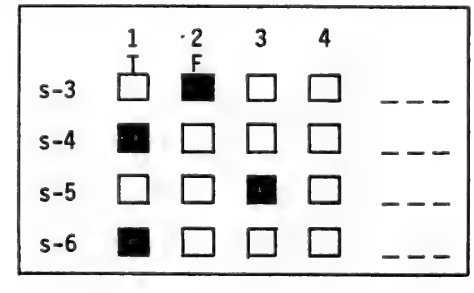
Each set of questions consists of two columns, each listing words, phrases or sentences. The task is to select the item in column B which is the best match for the item in column A that is being considered. Items in column B may be used once, more than once, or not at all. Specific instructions are given with each set of questions. Select the numbers identifying the answers and blacken the appropriate boxes on the answer sheet.

SAMPLE

In questions s-3 through s-6, match the name of the shipboard officer in column A by selecting from column B the name of the department in which the officer functions.

A	B
s-3. Damage Control Assistant	1. Operations Department
s-4. CIC Officer	2. Engineering Department
s-5. Disbursing Officer	3. Supply Department
s-6. Communications Officer	

Indicate in this way on the answer sheet:



Assignment 1

Introduction to Programming and the Fundamental Concepts and Structure of BASIC, Including Constants, Variables, Expressions, Arithmetic Operations, and Input/Output

Textbook Assignment: pages 1-1 through 4-6

Learning Objective: Differentiate between the problem solving steps used in programming.

1-1. Which of the following is the first and most important step in problem solving?

1. Preparing a flowchart of the problem solution
2. Developing a thorough understanding of the problem
3. Coding the program
4. Documenting the program

1-2. Once the problem definition is written, what is the next step in the problem-solving process?

1. Coding the program
2. Constructing the flowchart
3. Completing documentation
4. Preparing test data

Learning Objective: Describe the purpose and characteristics of flowcharts. Recognize the flowcharting symbols and their meanings.

1-3. A system flowchart is used to define the major phases of processing and to

1. specify equipment to be used
2. show the sequence of operations in a program
3. show the path of data through the problem solution
4. specify record layouts

1-4. A programming flowchart is used for which of the following purposes?

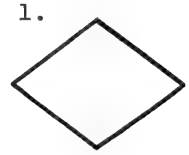
1. To represent the path of data through a problem solution
2. To represent the sequence of operations in a program
3. Both 1 and 2 above
4. To specify operator instructions

1-5. Who usually constructs the programming flowchart?

1. The customer
2. The data base analyst
3. The system analyst
4. The programmer

1-6. Which of the following flowcharting symbols would be used to indicate the testing of a relational condition?

1.

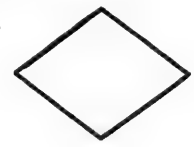
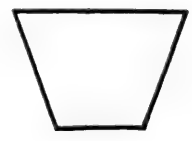


3.

2.

4.

TO ANSWER QUESTIONS 1-7 THROUGH 1-9, SELECT FROM COLUMN B THE SYMBOL THAT MATCHES THE OPERATION LISTED IN COLUMN A. ALL RESPONSES WILL NOT BE USED.

A. OPERATIONS	B. SYMBOLS
1-7. Compute average scores	1. 
1-8. Read data file	
1-9. If A > B THEN	2. 
	3. 
	4. 

1-10. Arrows are used to represent the flow direction on a flowchart. What is the normal direction of flow, if any?

1. Left to right only
2. Top to bottom only
3. Left to right and top to bottom
4. There is no normal direction

Learning Objective: Differentiate between the types of programming instructions and languages.

1-11. The symbols in a flowchart represent the instruction(s) to be performed by the computer. What type of instruction allows comparison between data?

1. Arithmetic
2. Logic
3. Input/output
4. Unconditional transfer

1-12. An instruction which causes a change in the normal sequence of execution is called what type of instruction?

1. Logic
2. Transfer of control
3. Decision
4. Input/output

1-13. A programming language that is written in a code the computer understands without being translated is known as

1. symbolic language
2. machine language
3. procedure-oriented language
4. macro language

1-14. A programming language that is oriented toward a specific class of processing problems is known as

1. machine language
2. symbolic language
3. procedure-oriented language
4. macro language

1-15. Which of the following is NOT an advantage realized as a result of using a procedure-oriented language?

1. It is easier to learn
2. It is compatible with a variety of computers
3. It is easier to maintain
4. It uses less memory space

Learning Objective: Describe the final steps in the programming process.

1-16. The process of manually tracing data through typical program paths is known as

1. desk-checking
2. debugging
3. testing
4. verifying

1-17. The process of determining that valid and invalid data can be processed correctly and that the output is correct is known as

1. debugging
2. implementing
3. testing
4. documenting

1-18. The process of collecting, organizing, storing, and maintaining a history of a program and documents associated with it is known as

1. implementation
2. data collection
3. systems analysis
4. documentation

1-19. The method of processing a job using both the old program and the new program is known as

1. multiprocessing
2. auxiliary processing
3. parallel processing
4. testing

Learning Objective: Describe the fundamental concepts, structure, and syntax of BASIC.

1-20. Which of the following is the primary purpose for which BASIC was originally designed?

1. To be an easy-to-learn language
2. To be a scientific language
3. To be a general-purpose language
4. To be a file-processing language

1-21. Which of the following parts of a BASIC statement is used to indicate to the computer the sequence of the instructions?

1. Descriptive information
2. Keyword
3. Line number
4. Predefined function

1-22. What part of an instruction is used to tell the computer what function is to be performed?

1. Keyword
2. Operand
3. Interpreter
4. Line number

1-23. Which of the following is a valid BASIC line number?

1. 99.9
2. +85
3. 595
4. 7/8

1-24. In BASIC, which of the following symbols would be used to indicate a multiplication operation?

1. **
2. X
3. /
4. *

1-25. Which of the following numbers is valid in BASIC?

1. 3,245
2. 1/10
3. 9.76
4. $12.00

1-26. How would the number 34567890000 be represented using scientific notation in BASIC?

1. 3.456789E10
2. 3.456789E6
3. 3.46E10
4. 3.5E10

1-27. Some mathematical operations are so frequently used they have been incorporated into BASIC. They are known as

1. syntax functions
2. macro functions
3. fundamental functions
4. predefined functions

1-28. If you entered two statements with the same line number, which of the following conditions would prevail?

1. You would get an error message
2. The first statement with the duplicate statement number would be executed
3. The computer would assign another statement number to the duplicate statement
4. The second statement would replace the first one

1-29. Which of the following is the primary reason for incrementing line numbers by 10?

1. It is easier to keep track of the numbers used
2. It allows you to insert statements later without renumbering
3. It is required by the BASIC language
4. It makes it easier to reference statements to be changed

QUESTIONS 1-30 THROUGH 1-33 ARE TO BE JUDGED TRUE OR FALSE.

1-30. When you are keying in a BASIC program and you omit a statement, you can enter it later as long as you assign the statement a line number that will cause it to be placed in the proper sequence.

1. True
2. False

1-31. There must be a space between each character in a BASIC statement.

1. True
2. False

1-32. Assume you make several mistakes while keying in a BASIC statement. You can correct them by depressing the backspace key once for each character you wish to erase or replace.

1. True
2. False

1-33. Logic errors are detected in a BASIC program by using the EDIT function.

1. True
2. False

1-34. Which of the following commands could be used to display a statement that has already been entered into the computer?

1. DELETE
2. INSERT
3. EDIT/RECALL
4. RETURN

1-35. Which of the following keywords is used to include documentation and/or programmer comments in a program?

1. COMMENT
2. ANNOTATION
3. REMARK
4. DOCUMENTATION

Learning Objective: Describe the uses of END and PRINT statements in BASIC programs and recognize their syntax.

1-36. What keyword indicates to the BASIC compiler that there are no more statements to be translated and terminates execution of a program?

1. PRINT
2. LET
3. END
4. STOP

1-37. In BASIC, the predefined format for printed output is referred to as what type of spacing?

1. Packed
2. Standard
3. Predefined
4. Zoned

1-38. Which of the following PRINT statements would print the value of a variable named Z?

1. 10 PRINT Z
2. 10 PRINT "Z="
3. 10 PRINT "Z"
4. 10 PRINT (Z)

1-39. Which of the following PRINT statements would print the values for the variables X, Y, and Z?

1. 20 PRINT XYZ
2. 20 PRINT "X,Y,Z"
3. 20 PRINT X+Y+Z
4. 20 PRINT X,Y,Z

1-40. GIVEN: 10 PRINT (52+64-32)*4

What would be printed?

1. (52+64-32)*4
2. 52+64-32*4
3. 336
4. (52+64-32)*4=336

1-41. To print a message, the message must be enclosed in what punctuation?

1. Commas
2. Semicolons
3. Parentheses
4. Quotation marks

1-42. If a blank PRINT statement is included in a program, what effect, if any, will it have on the output?

1. It will cause a blank line in the output
2. It will cause an error message
3. It will serve as filler in case you want to add a PRINT statement later
4. It will have no effect, blank PRINT statements are ignored

1-43. In a PRINT statement, what punctuation marks are used as separators to cause standard spacing?

1. Semicolons
2. Commas
3. Quotation marks
4. Parentheses

1-44. In a PRINT statement, what punctuation marks are used as separators to cause packed spacing?

1. Commas
2. Quotation marks
3. Parentheses
4. Semicolons

1-45. What is the purpose of using a comma at the end of a PRINT statement?

1. To cause a message to be printed exactly as it appears in the program
2. To cause a blank line in the printed output
3. To cause a blank field in the printed output
4. To allow the information in the following PRINT statement to be printed on the same line

1-46. When a comma is used at the end of a PRINT statement, where will the data in the following PRINT statement begin printing?

1. In the next available print zone
2. On the next print line
3. Immediately following the last character in the preceding print line
4. One space after the last character in the preceding print line

1-47. Which of the following effects will a semicolon in a PRINT statement have on the printed output?

1. Numeric data will always have two blank spaces between data elements
2. Messages will always be printed together
3. Both 1 and 2 above
4. Messages and numeric data will have two blank spaces between data elements

1-48. Where, if anywhere, are the results of computations done in PRINT statements stored?

1. They are stored in the program
2. They are stored in the computer's memory
3. They are stored on tape for later use
4. They are not stored, only printed

Learning Objective: Identify the symbols used for arithmetic operations. Write BASIC statements to solve arithmetic expressions.

1-49. What BASIC keyword is used to assign a value to a variable name or to instruct the computer to do a computation and assign the result to a variable name?

1. LET
2. PRINT
3. RESTORE
4. DEFINE

1-50. Which of the following best describes a LET statement?

1. It is a statement of algebraic equality
2. It is known as an assignment statement
3. Its values cannot be stored in a computer's memory
4. It is a value represented by algebraic symbols

1-51. Which of the following arithmetic expressions would be used to perform multiplication in BASIC?

1. XY
2. X·Y
3. XxY
4. X*Y

1-52. In BASIC, if you forget to include the appropriate arithmetic operator in an arithmetic expression, which of the following conditions occurs?

1. The computer ignores the statement
2. The computer automatically assumes multiplication
3. The computer gives you an error condition
4. The computer uses the operator from the previous arithmetic expression

1-53. According to the precedence rule, which of the following arithmetic operations will the computer perform first?

1. **
2. *
3. /
4. +

1-54. GIVEN: 20 LET M=8/4*3

What would be the result?

1. .75
2. 6
3. 9
4. 96

1-55. What punctuation marks are used to alter the order of precedence in arithmetic expressions?

1. Commas
2. Parentheses
3. Semicolons
4. Quotation marks

1-56. Which of the following arithmetic expressions is properly coded to solve $B=\frac{M}{X+Y}$ in BASIC?

1. LET B=M/X+Y
2. LET B=(M/X+Y)
3. LET B=(M/X)+Y
4. LET B=M/(X+Y)

1-57. What is the order of performance of arithmetic operators when parentheses are inside parentheses?

1. Left to right
2. Inside pair performed first
3. Exponentiation is always done first
4. Right to left

Learning Objective: Define and recognize constants and variables.

1-58. What is the term used to refer to a location in memory whose value may change during program execution?

1. Constant
2. String
3. Variable
4. Operator

1-59. Constants have which of the following characteristics?

1. They may contain character strings only
2. They may contain numeric data only
3. Their values remain the same during program execution
4. Their values may change during program execution

1-60. Which of the following is a properly coded numeric-variable name?

1. M$
2. 1M
3. XY
4. A1

1-61. Which of the following variable names could be used to represent the data, "FRED MCGEE", "DPC"?

1. N1, R1
2. N$, RA
3. NA, R$
4. N$, R$

1-62. Which of the following describes numeric-variable names?

1. A single numeric digit (0-9)
2. Any two alphabetic characters (A-Z)
3. A single alphabetic character (A-Z), or one alphabetic character (A-Z), followed by a single numeric digit (0-9)
4. A single numeric digit (0-9) followed by one alphabetic character (A-Z)

Learning Objective: Code statements to introduce data into a computer system.

1-63. What keyword is used to access data stored in a program and assign it a variable name?

1. DATA
2. READ
3. INPUT
4. RESTORE

1-64. Data stored in a data list in the computer's memory is specified by what keyword?

1. DATA
2. RESTORE
3. LET
4. LOAD

1-65. In what order are the values in data lists assigned to variable names in READ statements?

1. Randomly
2. Arbitrarily
3. Consecutively
4. Alphabetically

1-66. Which of the following conditions occurs when an excess of data in DATA statements is encountered?

1. The computer reassigns variable names beginning with the first one
2. The computer gives an error message
3. The computer terminates the program
4. The computer ignores the excess data

1-67. What punctuation marks are used to separate variables in READ statements and values in DATA statements?

1. Commas
2. Semicolons
3. Colons
4. Parentheses

1-68. GIVEN:
```
10 READ M,G,S,Y
20 DATA 475,16,42,20
```

What value would be assigned to the variable G?

1. 16
2. 20
3. 42
4. 475

1-69. What keyword is used to reset the pointer to the top of the data list in DATA statements?

1. RESET
2. RESTORE
3. LOAD
4. RETURN

1-70. What keyword is used to introduce data into a program during program execution?

1. READ
2. RESTORE
3. LOAD
4. INPUT

1-71. What punctuation is used to separate variable names specified in INPUT statements?

1. Semicolon
2. Question mark
3. Colon
4. Comma

```
10 READ P,Q,R
20 RESTORE
30 READ X,Y,Z
40 INPUT A,B
50 DATA 9,7,5
60 DATA 3,1
99 END
```

Figure 1A.--Program.

IN ANSWERING QUESTIONS 1-72 THROUGH 1-74, REFER TO FIGURE 1A.

1-72. After execution of the READ statement in line 10, at what value would the pointer be positioned?

1. 1
2. 5
3. 3
4. 9

1-73. What values, if any, would be assigned to the variables X, Y, and Z by the READ statement, line 30?

1. 9,7,5 would be assigned to X,Y, and Z respectively
2. 5,3,1 would be assigned to X,Y, and Z respectively
3. 3 and 1 would be assigned to X and Y respectively, no value would be assigned to Z
4. The computer would give an "INSUFFICIENT DATA" error message

1-74. What values would be assigned the variable names in the INPUT statement in line 40?

1. Any two alphabetic values entered by the user
2. Any two numeric values entered by the user
3. The first and second values in the DATA statement in line 50
4. The first and second values in the DATA statement in line 60

1-75. If you are entering data via an INPUT statement and you enter too few values, which of the following conditions occurs?

1. The computer processes the values entered
2. The computer gives you an "INSUFFICIENT DATA" error message
3. The computer responds with question marks until the specified number of values are entered
4. The program terminates

Assignment 2

Transfer of Control Statements, Loops, Functions, Subroutines, Arrays, Printed Output, and File Processing Concepts

Textbook Assignment: pages 4-6 through 7-14 and figure 7-2

Learning Objective: Select and use transfer of control statements to alter the normal sequence of program execution and to control program loops.

2-1. In BASIC, a transfer of control instruction which alters the normal sequence of program execution based on the evaluation of a relationship is known as which of the following?

1. Conditional
2. Unconditional
3. Assignment
4. Subroutine

2-2. Which of the following keywords is used to unconditionally alter the normal sequence of program execution?

1. IF-THEN
2. ON-GOTO
3. COMPUTED GOTO
4. GOTO

2-3. Which of the following terms is used to refer to any sequence of instructions that is to be repeated some specified number of times or until a particular condition is met?

1. Loop
2. Subroutine
3. Branching
4. Cycling

2-4. What is the maximum number of conditions that result from the evaluation of a relational IF-THEN statement?

1. One
2. Two
3. Three
4. Four

2-5. The relational IF-THEN statement transfers control only if the state of the relationship tested proves to be which of the following?

1. True
2. False
3. Equal
4. Unequal

2-6. Which of the following statements would result in a branch to line 90 when X equals 70?

1. 40 IF END = 70 THEN 90
2. 40 IF X = 70 THEN 90
3. 40 ON X = 70 THEN 90
4. 40 ON X = 70 GOTO 90

2-7. Which of the following statements could be used to transfer control to line 200 when the loop has been executed 15 times?

1. 30 GOTO 200 WHEN C=15
2. 30 WHEN C=15 then 200
3. 30 IF C=15 THEN 200
4. 30 IF C EQUALS 15 THEN 200

2-8. A loop can be controlled by using a numeric variable as a counter and testing to determine when the counter reaches a predetermined number. Which of the following statements could be used to accumulate a count?

1. 80 ADD 1 TO C
2. 80 LET C EQUAL C PLUS 1
3. 80 LET C+1=C
4. 80 LET C=C+1

2-9. Before you use a variable as a counter, it is good practice to initialize it to be sure it

1. is defined by the program
2. contains the value you want it to have
3. contains a one
4. can be referenced as a counter

2-10. GIVEN:
10 ON X/10 GOTO 60,90,110,110

Which of the following values of X would cause control to be transferred to line 90?

1. X=2 only
2. X=20 only
3. X≥2 and <3
4. X≥20 and <30

IN ANSWERING QUESTIONS 2-11 THROUGH 2-14, REFER TO FIGURE 4-2 IN THE TEXT.

2-11. If you entered a Fahrenheit temperature of 220 and a 1, which set of statements would be executed following statements 60, 90, and 100?

1. 110,120,170,180
2. 110,130,140,170,180
3. 130,140,170,180
4. 130,140,150,170,180

2-12. Line 130 could be executed if which of the following conditions exists?

1. A Celsius temperature converted to Fahrenheit is equal to or greater than 100°F
2. A 2 is entered at line 50
3. A Fahrenheit temperature converted to Celsius is greater than 100°C
4. A Celsius temperature greater than 100° is entered at line 20

2-13. What value(s) input at line 180 will cause line 10 to be executed?

1. Q$
2. "Q$"
3. Both 1 and 2 above
4. Y

2-14. Under which, if any, of the following conditions will line 70 be executed?

1. When an integer value other than 1 or 2 is entered at line 50
2. When a Fahrenheit temperature and a 2 are entered
3. When a Celsius temperature and a 1 are entered
4. None of the above

Learning Objective: Define the structure and rules of FOR-NEXT loops.

2-15. Which of the following is the name given to the variable whose value is used to determine when a loop has been executed the specified number of times?

1. String-variable
2. Loop variable
3. Running variable
4. Control variable

2-16. Which of the following could be the first statement of a FOR-NEXT loop?

1. FOR X = 20
2. FOR X = 1 TO 20
3. FOR X = 1 NEXT 20
4. NEXT 20

2-17. Unless otherwise specified, the value of the running variable is increased by what value each time a FOR-NEXT loop is executed?

1. One
2. Two
3. Three
4. Four

2-18. GIVEN:
```
40 FOR A = 1 TO 20
50 PRINT A**2
60 NEXT A
```

What total number of times will this loop be executed?

1. 1
2. 19
3. 20
4. 21

2-19. What keyword(s) would be used to specify a running variable's value is to be incremented by some value other than one?

1. FOR-TO
2. FOR-NEXT
3. STEP
4. NEXT

QUESTIONS 2-20 THROUGH 2-23 ARE TO BE JUDGED TRUE OR FALSE.

2-20. A FOR-NEXT loop must always begin with a FOR-TO statement and end with a NEXT statement.

1. True
2. False

2-21. A running variable cannot be used inside a loop.

1. True
2. False

2-22. The running variable name used in the FOR-TO statement does not have to be the same as the one used in the NEXT statement.

1. True
2. False

2-23. Control can be transferred out of a FOR-NEXT loop but not into one.

1. True
2. False

2-24. Under which of the following conditions would a FOR-NEXT loop NOT be executed?

1. The initial and final values of the running variable are equal and the step size is zero
2. The final value of the running variable is less than the initial value and the step size is negative
3. The final value of the running variable is greater than the initial value and the step size is positive
4. Each of the above

2-25. A loop within a loop is known as what kind of loop?

1. DO
2. Computed GOTO
3. FOR-NEXT
4. Nested

Learning Objective: Define and use arrays.

2-26. What keyword is used to define the size of an array?

1. DATA
2. DEFINE
3. LET
4. DIMENSION

2-27. A subscripted variable is used to do which of the following?

1. Cause a branch to a subroutine
2. Return control from a subroutine to the main program
3. Indicate a data element's relative position in an array
4. Define the number of memory spaces needed for an array

2-28. Suppose you wanted to reference position five in a one dimensional array, which of the following terms could be used to accomplish this?

1. DIM(4,1)
2. D(5)
3. D(4,1)
4. DIM(5)

2-29. Which of the following DIM statements is properly coded to reserve 150 spaces in memory for a two-dimensional array with 15 columns and 10 rows?

1. 10 DIM (15,10)
2. 10 DIM A(15,10)
3. 10 DIM A(10,15)
4. 10 DIM (10,15)

2-30. Memory space for what total number of data elements is automatically reserved for a one-dimensional array?

1. 11
2. 12
3. 121
4. 133

2-31. Which of the following BASIC statements correctly references the data in row 12, column 10 of a matrix containing numeric data?

1. 10 LET X=Y(12,10)
2. 10 LET X$=Y$(12,10)
3. 10 LET X=Y(10,12)
4. 10 LET X$=Y$(10,12)

2-32. Which of the following array names correctly identifies an array containing string data?

1. AB
2. D1
3. 1A$
4. A$

Learning Objective: Describe functions and subroutines, including their syntax and keywords used.

2-33. GIVEN: 60 PRINT ABS (-10)

What is the name given to the to the number in parentheses?

1. Expression
2. Argument
3. Function
4. Value

2-34. If there is no predefined function to perform the computation you want to do, you may define your own by using what keyword?

1. FND
2. DEF
3. DIM
4. FN

2-35. Which of the following is a correctly-coded string function name?

1. FN$
2. FNA
3. FNA$
4. FN1$

2-36. At what point is the expression in a function definition statement evaluated?

1. When the program is compiled
2. When the defined function is referenced
3. When the function definition statement is encountered
4. When the argument is defined

2-37. A function may be defined a maximum of how many times in a program?

1. One
2. Two
3. Three
4. Four

2-38. What is the maximum number of numeric functions that may be defined in a single program?

1. 1
2. 26
3. 32
4. 36

2-39. A small program within another program is known as which of the following?

1. Loop
2. Nested loop
3. DO loop
4. Subroutine

2-40. What keyword(s) is/are used to transfer control to a subroutine?

1. GOTO
2. ON-GOTO
3. GOSUB
4. IF-THEN

2-41. What keyword is used to transfer control from a subroutine back to the main program?

1. RESET
2. RESTORE
3. RETURN
4. STOP

2-42. When control is transferred from a subroutine back to the main program, to what statement does control return?

1. The statement immediately following the RETURN statement
2. The statement immediately preceding the GOSUB statement
3. The last statement of the subroutine
4. The statement immediately following the GOSUB statement

2-43. Which of the following descriptions applies to the use of a STOP statement?

1. It must be the last statement of a subroutine
2. It must be the last statement in a program
3. It may be placed anywhere in a program
4. It may be used in place of an END statement

Learning Objective: Code statements to format printed output.

2-44. The number or expression in parentheses following a TAB function is used to indicate the

1. number of characters in that field
2. number of blank spaces to be left at the beginning of that field
3. position where printing is to begin
4. line number where the values for the variables are stored

2-45. If a print position specified in a TAB function has already been passed, where will the data print?

1. In the next print zone
2. In the next available print position
3. The printer will backspace to the specified print position and overprint
4. On the next print line in the specified print position

2-46. What punctuation marks are most commonly used as separators in PRINT statements containing TAB functions?

1. Colons
2. Semicolons
3. Commas
4. Parentheses

2-47. Assume you are to center a column heading with 8 characters over a data field with 16 characters. What total number of blank print positions would be to the left of the heading?

1. Six
2. Five
3. Three
4. Four

2-48. Which of the following entries must be included in a PRINT USING statement?

1. Line number of a format statement
2. Variables or expressions
3. Both 1 and 2 above
4. Literals and format control characters

QUESTIONS 2-49 THROUGH 2-52 ARE TO BE JUDGED TRUE OR FALSE.

2-49. Literals specified in format lines must be enclosed in quotation marks.

1. True
2. False

2-50. Format lines may contain a combination of literals and format control characters.

1. True
2. False

2-51. When you are printing only literals, they may be included in the PRINT USING statement.

1. True
2. False

2-52. Only string variables may be printed using the TAB function.

1. True
2. False

2-53. By using an output image you may print numeric-variables in which of the following ways?

1. In any location on a print line
2. Left justified
3. With leading zeros
4. Each of the above

2-54. To use an output image you must have a/an

1. FORMAT statement and a PRINT statement
2. FORMAT statement and a PRINT USING statement
3. IMAGE statement and a PRINT statement
4. IMAGE statement and a PRINT USING statement

2-55. GIVEN:
```
30 PRINT USING 50,B
50 %$#,###.##
```

If B equals 199.255, what will print?

1. $1,992.55
2. $ 199.25
3. $199.25
4. $199.26

Learning Objective: Describe files and file handling concepts, terminology, and processing.

2-56. A series of characters used to identify a program is known as a

1. master file
2. transaction file
3. data name
4. program name

2-57. What is the primary reason for assigning a name to a program?

1. To allow it to be stored independently from the data
2. So it can be used with different sets of data
3. So it can be referenced, loaded, and run repetitively
4. To allow it to be executed more than once

2-58. Data stored independently from a program on an auxiliary storage medium is known as a

1. field
2. file
3. record
4. zone

2-59. Which of the following is a specified area of a record used for a particular category of data?

1. Field
2. File
3. Character
4. Zone

2-60. An item of data with all its associated data is known by which of the following terms?

1. Field
2. File
3. Record
4. Zone

2-61. Which of the following types of statements accomplishes end-of-file processing and signals the computer the file is no longer needed?

1. WRITE
2. RESET
3. RESTORE
4. CLOSE

2-62. Which of the following types of statements names a data file and makes it available for processing?

1. WRITE
2. OPEN
3. CLOSE
4. SAVE

2-63. The process of posting transactions to a master file in order to reflect current data is know as

1. editing
2. validating
3. verifying
4. updating

IN ANSWERING QUESTIONS 2-64 THROUGH 2-68, REFER TO FIGURE 7-2 IN THE TEXT AND THE RELATED TEXT MATERIAL.

2-64. The program has what total number of loops?

1. Six
2. Five
3. Three
4. Four

2-65. Numeric-variable K is used for which of the following purposes?

1. To control the FOR-NEXT loops
2. To count the number of records updated
3. Both 1 and 2 above
4. To control reading of the master file

2-66. Which of the following is the purpose of blocks 7,8, and 9 on the flowchart?

1. Test for end of part numbers to be updated
2. Update on-hand quantity
3. Search array P for matching part number
4. Accumulate a total of parts data

2-67. If a part number is entered and not found in the master data, the program will do which of the following things?

1. Terminate with an error message
2. Print part # not found, enter part #
3. Add the part number to the master data
4. Assume end of data and go to the next segment of the program

2-68. Which of the following conditions would cause the parts data information to be printed on the items below stock level list?

1. If the on-hand quantity is less than the reorder-point quantity
2. If the on-hand quantity is equal to the reorder-point quantity
3. Both 1 and 2 above
4. If the on-hand quantity is greater than the reorder-point quantity

COURSE DISENROLLMENT

All study materials must be returned. On disenrolling, fill out only the upper part of this page and attach it to the inside front cover of the textbook for this course. Mail your study materials to the Naval Education and Training Program Development Center.

PRINT CLEARLY

NAVEDTRA NUMBER	COURSE TITLE
10079-2	OCC-ECC Introduction to Programming in BASIC

Name Last First Middle

Rank/Rate Designator Social Security Number

COURSE COMPLETION

Letters of satisfactory completion are issued only to personnel whose courses are administered by the Naval Education and Training Program Development Center. On completing the course, fill out the lower part of this page and enclose it with your last set of answer sheets. Be sure mailing addresses are complete. Mail to the Naval Education and Training Program Development Center.

PRINT CLEARLY

NAVEDTRA NUMBER	COURSE TITLE
10079-2	OCC-ECC Introduction to Programming in BASIC

Name

ZIP CODE

MY SERVICE RECORD IS HELD BY:

Activity

Address ZIP CODE

Signature of enrollee

A FINAL QUESTION: What did you think of this course? Of the text material used with the course? Comments and recommendations received from enrollees have been a major source of course improvement. You and your command are urged to submit your constructive criticisms and your recommendations. This tear-out form letter is provided for your convenience. Typewrite if possible, but legible handwriting is acceptable.

Date _____

From: _____
(RANK, RATE, CIVILIAN)

_____

_____ ZIP CODE _____

To: Naval Education and Training Program Development Center (Code 314)
Pensacola, Florida 32509-5000

Subj: NAVEDTRA 10079-2, OCC-ECC Introduction to Programming in BASIC; comments concerning

1. The following comments are hereby submitted:

(Fold along dotted line and staple or tape)

(Fold along dotted line and staple or tape)

DEPARTMENT OF THE NAVY

NAVAL EDUCATION AND TRAINING PROGRAM
DEVELOPMENT CENTER (Code 314)
PENSACOLA, FLORIDA 32509-5000

OFFICIAL BUSINESS
PENALTY FOR PRIVATE USE, $300

POSTAGE AND FEES PAID
NAVY DEPARTMENT
DoD-316

NAVAL EDUCATION AND TRAINING PROGRAM DEVELOPMENT CENTER
BUILDING 2435 (Code 314)
PENSACOLA, FLORIDA 32509-5000

PRINT OR TYPE

NAME _____ ADDRESS _____
Last First Middle Street/Ship/Unit/Division, etc.

City or FPO State Zip

RANK/RATE _____ SOC. SEC. NO. _____ DESIGNATOR _____ ASSIGNMENT NO. _____

☐ USN ☐ USNR ☐ ACTIVE ☐ INACTIVE OTHER (Specify) _____ DATE MAILED _____

SCORE

	1 T	2 F	3	4			1 T	2 F	3	4			1 T	2 F	3	4	
1	☐	☐	☐	☐	_____	26	☐	☐	☐	☐	_____	51	☐	☐	☐	☐	_____
2	☐	☐	☐	☐	_____	27	☐	☐	☐	☐	_____	52	☐	☐	☐	☐	_____
3	☐	☐	☐	☐	_____	28	☐	☐	☐	☐	_____	53	☐	☐	☐	☐	_____
4	☐	☐	☐	☐	_____	29	☐	☐	☐	☐	_____	54	☐	☐	☐	☐	_____
5	☐	☐	☐	☐	_____	30	☐	☐	☐	☐	_____	55	☐	☐	☐	☐	_____
6	☐	☐	☐	☐	_____	31	☐	☐	☐	☐	_____	56	☐	☐	☐	☐	_____
7	☐	☐	☐	☐	_____	32	☐	☐	☐	☐	_____	57	☐	☐	☐	☐	_____
8	☐	☐	☐	☐	_____	33	☐	☐	☐	☐	_____	58	☐	☐	☐	☐	_____
9	☐	☐	☐	☐	_____	34	☐	☐	☐	☐	_____	59	☐	☐	☐	☐	_____
10	☐	☐	☐	☐	_____	35	☐	☐	☐	☐	_____	60	☐	☐	☐	☐	_____
11	☐	☐	☐	☐	_____	36	☐	☐	☐	☐	_____	61	☐	☐	☐	☐	_____
12	☐	☐	☐	☐	_____	37	☐	☐	☐	☐	_____	62	☐	☐	☐	☐	_____
13	☐	☐	☐	☐	_____	38	☐	☐	☐	☐	_____	63	☐	☐	☐	☐	_____
14	☐	☐	☐	☐	_____	39	☐	☐	☐	☐	_____	64	☐	☐	☐	☐	_____
15	☐	☐	☐	☐	_____	40	☐	☐	☐	☐	_____	65	☐	☐	☐	☐	_____
16	☐	☐	☐	☐	_____	41	☐	☐	☐	☐	_____	66	☐	☐	☐	☐	_____
17	☐	☐	☐	☐	_____	42	☐	☐	☐	☐	_____	67	☐	☐	☐	☐	_____
18	☐	☐	☐	☐	_____	43	☐	☐	☐	☐	_____	68	☐	☐	☐	☐	_____
19	☐	☐	☐	☐	_____	44	☐	☐	☐	☐	_____	69	☐	☐	☐	☐	_____
20	☐	☐	☐	☐	_____	45	☐	☐	☐	☐	_____	70	☐	☐	☐	☐	_____
21	☐	☐	☐	☐	_____	46	☐	☐	☐	☐	_____	71	☐	☐	☐	☐	_____
22	☐	☐	☐	☐	_____	47	☐	☐	☐	☐	_____	72	☐	☐	☐	☐	_____
23	☐	☐	☐	☐	_____	48	☐	☐	☐	☐	_____	73	☐	☐	☐	☐	_____
24	☐	☐	☐	☐	_____	49	☐	☐	☐	☐	_____	74	☐	☐	☐	☐	_____
25	☐	☐	☐	☐	_____	50	☐	☐	☐	☐	_____	75	☐	☐	☐	☐	_____

PRINT OR TYPE

NAME _____ ADDRESS _____
Last First Middle Street/Ship/Unit/Division, etc.

City or FPO State Zip

RANK/RATE _____ SOC. SEC. NO. _____ DESIGNATOR _____ ASSIGNMENT NO. _____

☐ USN ☐ USNR ☐ ACTIVE ☐ INACTIVE OTHER (Specify) _____ DATE MAILED _____

SCORE

	1 (T)	2 (F)	3	4		1 (T)	2 (F)	3	4		1 (T)	2 (F)	3	4	
1	☐	☐	☐	☐	_____	26	☐	☐	☐	☐	_____	51	☐	☐	☐	☐	_____
2	☐	☐	☐	☐	_____	27	☐	☐	☐	☐	_____	52	☐	☐	☐	☐	_____
3	☐	☐	☐	☐	_____	28	☐	☐	☐	☐	_____	53	☐	☐	☐	☐	_____
4	☐	☐	☐	☐	_____	29	☐	☐	☐	☐	_____	54	☐	☐	☐	☐	_____
5	☐	☐	☐	☐	_____	30	☐	☐	☐	☐	_____	55	☐	☐	☐	☐	_____
6	☐	☐	☐	☐	_____	31	☐	☐	☐	☐	_____	56	☐	☐	☐	☐	_____
7	☐	☐	☐	☐	_____	32	☐	☐	☐	☐	_____	57	☐	☐	☐	☐	_____
8	☐	☐	☐	☐	_____	33	☐	☐	☐	☐	_____	58	☐	☐	☐	☐	_____
9	☐	☐	☐	☐	_____	34	☐	☐	☐	☐	_____	59	☐	☐	☐	☐	_____
10	☐	☐	☐	☐	_____	35	☐	☐	☐	☐	_____	60	☐	☐	☐	☐	_____
11	☐	☐	☐	☐	_____	36	☐	☐	☐	☐	_____	61	☐	☐	☐	☐	_____
12	☐	☐	☐	☐	_____	37	☐	☐	☐	☐	_____	62	☐	☐	☐	☐	_____
13	☐	☐	☐	☐	_____	38	☐	☐	☐	☐	_____	63	☐	☐	☐	☐	_____
14	☐	☐	☐	☐	_____	39	☐	☐	☐	☐	_____	64	☐	☐	☐	☐	_____
15	☐	☐	☐	☐	_____	40	☐	☐	☐	☐	_____	65	☐	☐	☐	☐	_____
16	☐	☐	☐	☐	_____	41	☐	☐	☐	☐	_____	66	☐	☐	☐	☐	_____
17	☐	☐	☐	☐	_____	42	☐	☐	☐	☐	_____	67	☐	☐	☐	☐	_____
18	☐	☐	☐	☐	_____	43	☐	☐	☐	☐	_____	68	☐	☐	☐	☐	_____
19	☐	☐	☐	☐	_____	44	☐	☐	☐	☐	_____	69	☐	☐	☐	☐	_____
20	☐	☐	☐	☐	_____	45	☐	☐	☐	☐	_____	70	☐	☐	☐	☐	_____
21	☐	☐	☐	☐	_____	46	☐	☐	☐	☐	_____	71	☐	☐	☐	☐	_____
22	☐	☐	☐	☐	_____	47	☐	☐	☐	☐	_____	72	☐	☐	☐	☐	_____
23	☐	☐	☐	☐	_____	48	☐	☐	☐	☐	_____	73	☐	☐	☐	☐	_____
24	☐	☐	☐	☐	_____	49	☐	☐	☐	☐	_____	74	☐	☐	☐	☐	_____
25	☐	☐	☐	☐	_____	50	☐	☐	☐	☐	_____	75	☐	☐	☐	☐	_____

PRINT OR TYPE

NAME _____ ADDRESS _____
Last First Middle Street/Ship/Unit/Division, etc.

City or FPO State Zip

RANK/RATE _____ SOC. SEC. NO. _____ DESIGNATOR _____ ASSIGNMENT NO. _____

☐ USN ☐ USNR ☐ ACTIVE ☐ INACTIVE OTHER (Specify) _____ DATE MAILED _____

SCORE

	1 T	2 F	3	4			1 T	2 F	3	4			1 T	2 F	3	4	
1	☐	☐	☐	☐	_____	26	☐	☐	☐	☐	_____	51	☐	☐	☐	☐	_____
2	☐	☐	☐	☐	_____	27	☐	☐	☐	☐	_____	52	☐	☐	☐	☐	_____
3	☐	☐	☐	☐	_____	28	☐	☐	☐	☐	_____	53	☐	☐	☐	☐	_____
4	☐	☐	☐	☐	_____	29	☐	☐	☐	☐	_____	54	☐	☐	☐	☐	_____
5	☐	☐	☐	☐	_____	30	☐	☐	☐	☐	_____	55	☐	☐	☐	☐	_____
6	☐	☐	☐	☐	_____	31	☐	☐	☐	☐	_____	56	☐	☐	☐	☐	_____
7	☐	☐	☐	☐	_____	32	☐	☐	☐	☐	_____	57	☐	☐	☐	☐	_____
8	☐	☐	☐	☐	_____	33	☐	☐	☐	☐	_____	58	☐	☐	☐	☐	_____
9	☐	☐	☐	☐	_____	34	☐	☐	☐	☐	_____	59	☐	☐	☐	☐	_____
10	☐	☐	☐	☐	_____	35	☐	☐	☐	☐	_____	60	☐	☐	☐	☐	_____
11	☐	☐	☐	☐	_____	36	☐	☐	☐	☐	_____	61	☐	☐	☐	☐	_____
12	☐	☐	☐	☐	_____	37	☐	☐	☐	☐	_____	62	☐	☐	☐	☐	_____
13	☐	☐	☐	☐	_____	38	☐	☐	☐	☐	_____	63	☐	☐	☐	☐	_____
14	☐	☐	☐	☐	_____	39	☐	☐	☐	☐	_____	64	☐	☐	☐	☐	_____
15	☐	☐	☐	☐	_____	40	☐	☐	☐	☐	_____	65	☐	☐	☐	☐	_____
16	☐	☐	☐	☐	_____	41	☐	☐	☐	☐	_____	66	☐	☐	☐	☐	_____
17	☐	☐	☐	☐	_____	42	☐	☐	☐	☐	_____	67	☐	☐	☐	☐	_____
18	☐	☐	☐	☐	_____	43	☐	☐	☐	☐	_____	68	☐	☐	☐	☐	_____
19	☐	☐	☐	☐	_____	44	☐	☐	☐	☐	_____	69	☐	☐	☐	☐	_____
20	☐	☐	☐	☐	_____	45	☐	☐	☐	☐	_____	70	☐	☐	☐	☐	_____
21	☐	☐	☐	☐	_____	46	☐	☐	☐	☐	_____	71	☐	☐	☐	☐	_____
22	☐	☐	☐	☐	_____	47	☐	☐	☐	☐	_____	72	☐	☐	☐	☐	_____
23	☐	☐	☐	☐	_____	48	☐	☐	☐	☐	_____	73	☐	☐	☐	☐	_____
24	☐	☐	☐	☐	_____	49	☐	☐	☐	☐	_____	74	☐	☐	☐	☐	_____
25	☐	☐	☐	☐	_____	50	☐	☐	☐	☐	_____	75	☐	☐	☐	☐	_____

PRINT OR TYPE

NAME _____ ADDRESS _____
Last First Middle Street/Ship/Unit/Division, etc.

_____
City or FPO State Zip

RANK/RATE _____ SOC. SEC. NO. _____ DESIGNATOR _____ ASSIGNMENT NO. _____

☐ USN ☐ USNR ☐ ACTIVE ☐ INACTIVE OTHER (Specify) _____ DATE MAILED _____

SCORE

	1 T	2 F	3	4			1 T	2 F	3	4			1 T	2 F	3	4	
1	☐	☐	☐	☐	_____	26	☐	☐	☐	☐	_____	51	☐	☐	☐	☐	_____
2	☐	☐	☐	☐	_____	27	☐	☐	☐	☐	_____	52	☐	☐	☐	☐	_____
3	☐	☐	☐	☐	_____	28	☐	☐	☐	☐	_____	53	☐	☐	☐	☐	_____
4	☐	☐	☐	☐	_____	29	☐	☐	☐	☐	_____	54	☐	☐	☐	☐	_____
5	☐	☐	☐	☐	_____	30	☐	☐	☐	☐	_____	55	☐	☐	☐	☐	_____
6	☐	☐	☐	☐	_____	31	☐	☐	☐	☐	_____	56	☐	☐	☐	☐	_____
7	☐	☐	☐	☐	_____	32	☐	☐	☐	☐	_____	57	☐	☐	☐	☐	_____
8	☐	☐	☐	☐	_____	33	☐	☐	☐	☐	_____	58	☐	☐	☐	☐	_____
9	☐	☐	☐	☐	_____	34	☐	☐	☐	☐	_____	59	☐	☐	☐	☐	_____
10	☐	☐	☐	☐	_____	35	☐	☐	☐	☐	_____	60	☐	☐	☐	☐	_____
11	☐	☐	☐	☐	_____	36	☐	☐	☐	☐	_____	61	☐	☐	☐	☐	_____
12	☐	☐	☐	☐	_____	37	☐	☐	☐	☐	_____	62	☐	☐	☐	☐	_____
13	☐	☐	☐	☐	_____	38	☐	☐	☐	☐	_____	63	☐	☐	☐	☐	_____
14	☐	☐	☐	☐	_____	39	☐	☐	☐	☐	_____	64	☐	☐	☐	☐	_____
15	☐	☐	☐	☐	_____	40	☐	☐	☐	☐	_____	65	☐	☐	☐	☐	_____
16	☐	☐	☐	☐	_____	41	☐	☐	☐	☐	_____	66	☐	☐	☐	☐	_____
17	☐	☐	☐	☐	_____	42	☐	☐	☐	☐	_____	67	☐	☐	☐	☐	_____
18	☐	☐	☐	☐	_____	43	☐	☐	☐	☐	_____	68	☐	☐	☐	☐	_____
19	☐	☐	☐	☐	_____	44	☐	☐	☐	☐	_____	69	☐	☐	☐	☐	_____
20	☐	☐	☐	☐	_____	45	☐	☐	☐	☐	_____	70	☐	☐	☐	☐	_____
21	☐	☐	☐	☐	_____	46	☐	☐	☐	☐	_____	71	☐	☐	☐	☐	_____
22	☐	☐	☐	☐	_____	47	☐	☐	☐	☐	_____	72	☐	☐	☐	☐	_____
23	☐	☐	☐	☐	_____	48	☐	☐	☐	☐	_____	73	☐	☐	☐	☐	_____
24	☐	☐	☐	☐	_____	49	☐	☐	☐	☐	_____	74	☐	☐	☐	☐	_____
25	☐	☐	☐	☐	_____	50	☐	☐	☐	☐	_____	75	☐	☐	☐	☐	_____